IMPLEMENTATION OF A LOWER-UPPER SYMMETRIC GAUSS-SEIDEL IMPLICIT SCHEME FOR A NAVIER-STOKES FLOW SOLVER

A Thesis

by

JERRY WILLIAM CARTER II

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2010

Major Subject: Aerospace Engineering

IMPLEMENTATION OF A LOWER-UPPER SYMMETRIC GAUSS-SEIDEL IMPLICIT SCHEME FOR A NAVIER STOKES FLOW SOLVER

A Thesis

by

JERRY WILLIAM CARTER II

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Paul Cizmas
Committee Members,	Leland Carlson
	Andrew Duggleby
Head of Department,	Dimitris Lagoudas

May 2010

Major Subject: Aerospace Engineering

ABSTRACT

Implementation of a Lower-Upper Symmetric
Gauss-Seidel Implicit Scheme
for a Navier Stokes Flow Solver. (May 2010)
Jerry William Carter II, B.S., Texas A&M University
Chair of Advisory Committee: Dr. Paul G. A. Cizmas

The field of Computational Fluid Dynamics (CFD) is in a continual state of advancement due to new numerical techniques, optimization of existing codes, and the increase in memory and processing speeds of computers. In this thesis, the solution technique for a pre-existing Navier-Stokes flow solver is adapted from an explicit Runge Kutta method to a Lower-Upper Symmetric Gauss-Seidel (LU-SGS) implicit time integration method. Explicit time integration methods were originally used in CFD codes because these methods require less memory. Information needed to advance the flow in time is localized to each grid point. These explicit methods are, however, restricted by small time step sizes due to stability criteria. In contrast, implicit methods are unaffected by large time step sizes but are restricted by memory requirements due to the complexities of unstructured grids. The implementation of LU-SGS performs grid re-ordering for unstructured meshes because of the coupling of grid points in the integration method's solution. The explicit and implicit flow solvers were tested for inviscid flows in incompressible, compressible, and transoinc flow regimes. The results found by comparing the implicit and explicit algorithms revealed a significant speed up in convergence to steady state by the LU-SGS method in terms of iteration number and CPU time per iteration.

ACKNOWLEDGMENTS

I would like to personally thank my advisor, committe chair, and friend, Dr. Paul Cizmas, for his continued support and encouragement. His passion for CFD has heavily directed the development of this research. I would also like to thank Dr. Leland Carlson, Dr. Andrew Duggleby, and Dr. Othon Rediniotis for their guidance and teachings during the course of this research. Their aid has greatly added to my understanding of fluid mechanics and computational techniques.

I must thank my friends and fellow graduate students: Tom Brenner, Forrest Carpenter, Greg Worley, Raymond Fontenont, Brian Freno, and Robert Brown. A special thanks to my friend, roommate, and collaborator, David Liliedahl. All of their friendship has been prevalent in the process of this research. A special thanks to the entire Aerospace Engineering department at Texas A&M University.

I cannot go without thanking my family. To Momma and Daddy for their continual love and encouragement. Also for the love from my siter, brother-in law, and nephews: Emily, Joe, Carter, and Caleb. I thank my new family, Staci, for her endless support and love. Finally, I thank God for His continual presence in my life.

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

A. Statement of Work

The objective of this present work is to increase the residual convergence history of a steady Navier-Stokes flow solver. In order to meet those objectives an implicit flow solver has been developed from a pre-existing explicit solver. The implicit technique has been adapted for the use on unstructured meshes.

B. Background

Computer technology has allowed for larger amounts of memory storage and increasing processing speeds. Still, high fidelity Navier-Stokes flow solvers obtain solutions over large periods of time on the order of hours, days, and weeks. The computational costs arise from numerical simulations occurring on complex geometries. Originally, Navier-Stokes solvers implemented explicit techniques to carry out simulations. Explicit solvers are limited by the physical time step computed to carry out temporal integration. The physical time step is dependent on several variables, some including: local Mach number, density, temperature, viscosity, and grid geometry. Computational power has led to the use of highly refined meshes. Minute structures in a mesh require smaller time steps in order to accurately capture the physics of the flow. If time steps are too large, calculations are advanced faster than the fluid time scales and leads to diverging solutions.

An unsteady Navier-Stokes explicit solver was previously developed for unstruc-

---

The journal model is *IEEE Transactions on Automatic Control.*

tured meshes. The intent of the code was for use of internal flows for turbomahinery that have very small scale features. The code was then adapted for external flows and used for aeroelastic analysis. Grid deformation is needed for these cases and plays a large role in the physics of a fluid. The flow conditions for turbomachinery and aeroelasticy require very small time scales to accurately predict details of the flow. Current test cases are being considered for the code that are not as restricted in time as previous simulations. The need to compute solutions with large time steps has brought another adaptation to the Navier-Stokes solver.

C. Flow Solver

The technique implemented into the flow solver is an implicit time integration scheme known as Lower-Upper Symmetric Gauss-Seidel (LUSGS). The scheme was originally developed for structured meshes but is presented for unstructured meshes with grid re-ordering. The governing equations are formulated with the Finite Volume Method (FVM) and are discretized over the control volumes defined by a given mesh. A new method of time discretization of the governing equations is presented. Assumptions by LUSGS approximate the flux Jacobian which leads to the linearization of the equations. Updating the flow at a point in the grid still requires information from surrounding nodes connected to the point of interest. The solution is broken up into two steps dependent on the lower and upper parts of the matrix formed by the implicit system. The scheme is invariant to the time step and can be advanced in time at any rate. The rate at which a solution is obtained is also increased with this method.

D. Original Contributions of the Present Work

- Development of a grid re-ordering method for unstructured meshes

- Adaptation of the flow solver with an implicit integration technique
- Validation of LUSGS flow solver

E. Outline of Thesis

Chapter II presents the aerodynamic model and the Reynolds-Averaged Navier-Stokes Equations. Chapter III develops the numerical techniques used to solve the governing equations. The equations are given for the FVM and then discretized spatially. The convective flux calculations are also given. The orginal explicit method is shown and the LUSGS implicit scheme is formulated. Chapter IV presents the results of the validation of the implicit flow solver for inviscid flows about NACA 0012 and NACA 0015 airfoils. Inviscid, transonic flow through a channel with a circular arc is also validated. Finally, results for the Generic Transport Wing (GTW) for a compressible flow regime are presented. The final chapter details the conclusions and future work.

CHAPTER II

PHYSICAL MODEL

The aerodynamic model presented in this chapter presents the governing fluid flow equations. The variables in the equations are decomposed and results into the Reynolds-Averaged Navier-Stokes model. The final part of the chapter nondimensionalizes the governing fluid flow equations.

A. Aerodynamic Model

The equations governing the physics of fluid flow comprise of the conservation of mass, linear momentum, and energy equations. The system formed by these five equations are collectively known as the Navier-Stokes equations. Under the assumptions of negligent body forces and heat sources, the Navier-Stokes equations describe the motion for a compressible, unsteady, heat conducting, viscous fluid and are written as

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = 0 \tag{2.1}$$

$$\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_i u_j}{\partial x_j} = -\frac{\partial p}{\partial x_j} + \frac{\partial \tau_{ij}}{\partial x_j} \tag{2.2}$$

$$\frac{\partial \rho E}{\partial t} + \frac{\partial \rho H u_j}{\partial x_j} = -\frac{\partial q_j}{\partial x_j} + \frac{\partial u_i \tau_{ij}}{\partial x_j} \tag{2.3}$$

where ρ, u_i, p, E, and H are the flow density, velocity, pressure, total energy per unit mass, and total enthalpy, respectively. The viscous stress tensor found in the momentum and energy balance is represented by a Newtonian viscous fluid, where

$$\tau_{ij} = \mu(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}) + \lambda \frac{\partial u_k}{\partial x_k}\delta_{ij} \tag{2.4}$$

The dynamic viscosity is μ and λ is the bulk viscosity. Dissipation due to viscosity must be non-negative, which requires

$$\lambda + \frac{2}{3}\mu \geq 0 \tag{2.5}$$

The viscous forces are to be zero when the fluid is at rest. Stokes hypothesis claims that

$$\lambda + \frac{2}{3}\mu = 0 \tag{2.6}$$

The heat flux is represented by q_j and is governed by Fourier's Law,

$$q_j = -k\frac{\partial T}{\partial x_j} \tag{2.7}$$

where T is the temperature and k is the thermal conductivity of the fluid. The equation of state relates the pressure, density, and temperature of a fluid.

$$p = \rho RT \tag{2.8}$$

The gas constant for air is represtented with R.

The total energy per unit mass, E, and total enthalpy, H, are decomposed as

$$E = e + \frac{1}{2}(u_i u_i) \tag{2.9}$$

$$H = h + \frac{1}{2}(u_i u_i), \tag{2.10}$$

where e and h represent the internal energy and internal enthalpy, respectively. For a calorically perfect gas, internal energy and internal enthalpy are functions of temperature:

$$e = c_v T \tag{2.11}$$

$$h = c_p T \tag{2.12}$$

where c_v is the specific heat constant at a constant vaolume and c_p is the specific heat constant at constant pressure. The specific heat constants are related to the gas constant R by

$$c_p - c_v = R, \tag{2.13}$$

and the specific heat ratio, γ, for air at standard conditions is defined as

$$\frac{c_p}{c_v} = \gamma \tag{2.14}$$

1. Reynolds-Averaged Navier-Stokes Equations

The modeling of a fluid may include the prediction of the small scale features of a fluid. The effects due to flow disturbances, model geometry, and increasing flow velocities can lead to turbulence within a viscous fluid flow. The characteristics of turbulence are comprised of highly varying scaled, unsteady fluctuations. The properties of the fluid are decomposed into a time averaged value and a fluctuating value in order to capture the turbulent effects in the flow. Using Reynolds decomposition method, a fluid property then becomes

$$q = \overline{q} + q^{'}, \tag{2.15}$$

where $\overline{q}$ is the time averaged value and $q^{'}$ is the fluctuation. The time averaged value $\overline{q}$ is defined as

$$\overline{q} = \frac{1}{\Delta t} \int_0^{\Delta t} q(x, \tau) d\tau. \tag{2.16}$$

Density fluctuations play a pivotal role in compressible flows and are eliminated from the averaged equations using Favre averaging. The Favre averaging technique uses density to weight a specific variable, which is defined as

$$q = \hat{q} + q'' \tag{2.17}$$

where

$$\hat{q} \equiv \frac{\overline{\rho q}}{\overline{\rho}}. \tag{2.18}$$

The density weighting method is applied to the velocity components, total energy per unit mass, and enthalpy. Reynolds averaging is applied to the remaining variables.

$$u_i = \hat{u}_i + u_i'', \quad E = \hat{E} + E'', \quad H = \hat{H} + H'' \tag{2.19}$$

$$\begin{aligned} \rho &= \overline{\rho} + \rho', \qquad p = \overline{p} + p', \\ \tau_{ij} &= \overline{\tau}_{ij} + \tau'_{ij}, \quad q_j = -k(\tfrac{\partial \overline{T}}{\partial x_j} + \tfrac{\partial T'}{\partial x_j}). \end{aligned} \tag{2.20}$$

Once the state variables are decomposed using Reynolds and Favre averaging techniques and substituted into the governing equations, Eqs. (2.1)-(2.3), the Reynolds-averaged Navier Stokes equations are obtained.

$$\frac{\partial \overline{\rho}}{\partial t} + \frac{\partial \overline{\rho}\hat{u}_j}{\partial x_j} = 0 \tag{2.21}$$

$$\frac{\partial \overline{\rho}\hat{u}_i}{\partial t} + \frac{\partial \overline{\rho}\hat{u}_i\hat{u}_j}{\partial x_j} = -\frac{\partial \overline{p}}{\partial x_i} + \frac{\partial \overline{\tau}_{ij}}{\partial x_j} - \frac{\partial \overline{\rho u_i'' u_j''}}{\partial x_j} \tag{2.22}$$

$$\frac{\partial \overline{\rho}\hat{E}}{\partial t} + \frac{\partial \overline{\rho}\hat{H}\hat{u}_j}{\partial x_j} = -\frac{\partial \overline{q}_j}{\partial x_j} + \frac{\partial \hat{u}_i\overline{\tau}_{ij}}{\partial x_j} - \frac{\partial \hat{u}_i\overline{\rho u_i'' u_j''}}{\partial x_j} - \frac{\partial \overline{\rho h'' u_j''}}{\partial x_j} \tag{2.23}$$

where

$$\overline{\tau}_{ij} = \mu\left(\frac{\partial \overline{u}_i}{\partial x_j} + \frac{\partial \overline{u}_j}{\partial x_i} - \frac{2}{3}\frac{\partial \overline{u}_k}{\partial x_k}\delta_{ij}\right) \tag{2.24}$$

$$\overline{q_j} = -k\frac{\partial \overline{T}}{\partial x_j} \tag{2.25}$$

The Reynolds-averaged Navier Stokes Equations (RANS) introduce several unknowns into the flow model. The new unknowns are commonly referred to as the Reynolds stresses, $-\overline{\rho u_i'' u_j''}$, and the Reynolds heat fluxes, $\overline{\rho h'' u_j''}$. A closure model is needed to solve the under-determined system of equations introduced by RANS.

2. Non-Dimensionalization

The RANS equations formulated in the previous section, Eqs (2.21)-(2.23), are dependent on the units used to measure the variables. The resulting scales of the physical numbers that arise from the choice of the units causes several of orders of magnitude difference between the dimensions of the variables. In order to alleviate computional costs dealing with wide ranges of numbers the variables are non-dimensionalized.

The variables used in RANS are non-dimensionalized by a characteristic length, L, the reference speed of sound, c_∞, the reference density, ρ_∞, and the reference dynamic viscosity, μ_∞.

$$\begin{array}{llll} x_i^* = \frac{x_i}{L} & t^* = \frac{tc_\infty}{L} & \rho^* = \frac{\overline{\rho}}{\rho_\infty} & u_i^* = \frac{\overline{u}_i}{c_\infty} \\ p^* = \frac{\overline{p}}{\rho_\infty c_\infty^2} & E^* = \frac{\hat{E}}{c_\infty^2} & H^* = \frac{\hat{H}}{c_\infty^2} & \mu^* = \frac{\mu}{\mu_\infty} \end{array} \tag{2.26}$$

In non-dimensional form, the Reynolds-averaged Navier-Stokes Equations are written as:

$$\frac{\partial \rho^*}{\partial t^*} + \frac{\partial \rho^* u_i^*}{\partial x_i^*} = 0 \tag{2.27}$$

$$\frac{\partial \rho^* u_i^*}{\partial t^*} + \frac{\partial \rho^* u_i^* u_j^*}{\partial x_j^*} = -\frac{\partial p^*}{\partial x_j^*} + \frac{1}{Re}\frac{\partial \tau_{ij}^*}{\partial x_j^*} \tag{2.28}$$

$$\frac{\partial \rho^* E^*}{\partial t^*} + \frac{\partial \rho^* H^* u_j^*}{\partial x_j^*} = -\frac{1}{PrRe}\frac{\partial q_j^*}{\partial x_j^*} + \frac{1}{Re}\frac{\partial u_i^* \tau_{ij}^*}{\partial x_j^*}. \tag{2.29}$$

The resulting non-dimensional Reynolds number, Prandtl number, and Mach number are based on the reference values and are defined as

$$Re = \frac{\rho_\infty c_\infty L}{\mu_\infty} \tag{2.30}$$

$$Pr = \frac{c_p \mu}{k} \tag{2.31}$$

$$M = \sqrt{\frac{\gamma p}{\rho}} \tag{2.32}$$

.

CHAPTER III

NUMERICAL METHOD

The beginning of this chapter spatially discretizes the governing equations found in the previous chapter. Later the explicit and implicit methods employed to integrate the discretized equations in time are formulated. A fourth-order Runge Kutta method was originally implemented. A Lower-Upper Symmetric Gauss-Seidel method has been added. Grid re-ordering concludes this chapter.

A. Flow Solver

The numerical methods implemented to discretize the Navier-Stokes equations are presented in this section. An integral form of the governing equations using the Finite Volume Method (FVM) is introduced. The FVM decomposes the flow domain into control volumes. Constant values are assumed for each state variable within a control volume and are stored at the each cell vertex.

Grid generation developed for the flow solver creates the geometry of the control volumes utilized by FVM. A cell vertex method was implemented in the code. The control volumes are defined using an unstructured, median dual mesh technique.

Once the grid is defined, The integral form of the Navier-Stokes equations are spatially discretized. The convective, diffusive, and source terms in the equations are computed and stored as the residual of one time step. The equations thereby are set up as ordinary differential equations in time and are left to be integrated.

1. Navier-Stokes Equations in Integral From

The Navier-Stokes equations are expressed in an integral form for the implementation of the FVM. The FVM breaks the domain into infinitesimial control volumes over

which the Navier-Stokes equations are solved. The values stored within each control volume are assumed to be constant in FVM. The Navier-Stokes equations, Eqs. (2.1)-(2.3), in integral form are

$$\frac{\partial}{\partial t}\int_{\Omega}\mathbf{Q}d\Omega+\oint_{S}\mathbf{F}\cdot\hat{\mathbf{n}}dS=\int_{\Omega}\mathbf{G}d\Omega. \tag{3.1}$$

The volume of the control volume is Ω and the surface of the control volume is defined by S. The state variables are stored in the vector $\mathbf{Q}$. The vector $\mathbf{Q}$ is comprised of the following conservative variables:

$$\mathbf{Q}=\begin{pmatrix}\rho\\ \rho u\\ \rho v\\ \rho w\\ \rho E\end{pmatrix}. \tag{3.2}$$

The flux vector $\mathbf{F}$ contains the convective, $\mathbf{F_c}$, and viscous, $\mathbf{F_v}$, components of the Navier-Stokes equations.

$$\mathbf{F}=\mathbf{F_c}+\mathbf{F_v} \tag{3.3}$$

The convective flux terms are

$$\mathbf{F_c}=\begin{pmatrix}\rho V\\ \rho uV+pn_x\\ \rho vV+pn_y\\ \rho wV+pn_z\\ \rho HV+pV_g\end{pmatrix} \tag{3.4}$$

where V is the contravariant velocity and V_g is the rotational velocity along the normal direction $\mathbf{n}$. Taking into account the relative velocity of the rotating frame,

the contravariant velocity is

$$V = (\mathbf{v} - \mathbf{r} \times \omega) \cdot \mathbf{n} \tag{3.5}$$

where ω is the angular velocity and $\mathbf{r}$ is the position vector. The rotational velocity component in the normal direction is

$$V_g = (\mathbf{r} \times \omega) \cdot \mathbf{n}. \tag{3.6}$$

In the presence of rotational frames of reference, the governing equations, Eq. (3.1, must account for source terms due to rotation. The vector $\mathbf{G}$ contains the rotational source terms.

$$\mathbf{G} = \begin{pmatrix} 0 \\ (\rho u \times \omega) \cdot \mathbf{e_x} \\ (\rho v \times \omega) \cdot \mathbf{e_y} \\ (\rho w \times \omega) \cdot \mathbf{e_z} \\ 0 \end{pmatrix} \tag{3.7}$$

The viscous flux terms are

$$\mathbf{F_v} = \begin{pmatrix} 0 \\ \tau_{xx} n_x + \tau_{xy} n_y + \tau_{xz} n_z \\ \tau_{yx} n_x + \tau_{yy} n_y + \tau_{yz} n_z \\ \tau_{zx} n_x + \tau_{zy} n_y + \tau_{zz} n_z \\ \Phi_x n_x + \Phi_y n_y + \Phi_z n_z \end{pmatrix}. \tag{3.8}$$

The work done by the viscous stresses and the heat conduction make up the term Φ. Using indicial notation, Φ is

$$\Phi_i = u_j \tau_{ij} + k \frac{\partial T}{\partial x_i} \tag{3.9}$$

The thermal conductivity coefficient in Eq. (3.9) is represented by k and is defined

as

$$k = c_p \frac{\mu}{Pr}. \tag{3.10}$$

The Prandtl number, Pr, is 0.72 for air. Viscous stresses are defined using Eq (2.4).

2. Spatial Discretization

The computational domain provides the geometry on which the Navier-Stokes equations are solved. The domain is made of nodes which are the vertex of the control volumes where the cell averaged state variables are stored [1]. A median dual mesh is employed to create the control volumes within the grid. The control volumes are defined by the midpoints along each edge connected to a vertex. Each midpoint is then connected to neighboring midpoints with an intermediate face centroid. The median dual mesh is used due to the scheme's ability to handle unstructured and hybrid meshes. Figure (1) illustrates the median dual mesh.

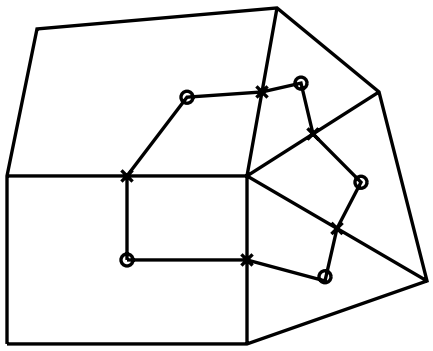

Fig. 1. Median dual mesh grid.

Unstructured meshes are desirable because of the lower computational fidelity

required to create. Structured meshes are employed in regions of the flow where large gradients occur, such as in boundary layers. The geometry created by an unstructured mesh is not as efficient and accurate in regions where there are significant gradients as structured meshes.

3. Navier-Stokes Equations in Semi-Discrete Form

The terms contained within the flux vector $\mathbf{F}$, and the rotation source term $\mathbf{G}$ are held constant for each time step. The state vectors, having been averaged over a cell, force the flux and source terms to no longer behave as continuous functions in space. These terms are then approximated discretely over a summation rather than continuously over an integral. The Navier-Stokes equations, Eq. (3.1), are then expressed for each control volume as ordinary differential equations.

Once the spatial approximations are performed the Navier-Stokes equations, Eq. (3.1), are simplified from a system of partial differential equations in space and time to ordinary differential equations in time alone. The discrete spatial method is advantageous because temporal approximations are independent of approximations made in space.

For the control volume at node i , the average value of the state vector is defined as

$$\mathbf{q}_\mathrm{i} \equiv \frac{1}{\Omega_\mathrm{i}} \int_{\Omega_\mathrm{i}} \mathbf{Q} d\Omega \tag{3.11}$$

where Ω_i is the volume of the cell. The average value over the volume for a source term at cell i is approximated by

$$\mathbf{g}_\mathrm{i} \approx \frac{1}{\Omega_\mathrm{i}} \int_{\Omega_\mathrm{i}} \mathbf{G} d\Omega. \tag{3.12}$$

The discretization of the flux vector $\mathbf{F}$ requires the surface integral to be approximated

as a summation over all the faces of a control volume. Let $\mathbf{n}_{\mathrm{ij}}$ and A_{ij} be the normal vector and area of the cell face between node i and j. Then the surface integral of the fluxes for cell i can be represented as

$$\oint_{d\Omega_{\mathrm{i}}} \mathbf{F} \cdot \hat{\mathbf{n}} d\mathrm{S} \approx \sum_{\mathrm{j}=1}^{\mathrm{nf(i)}} \mathbf{f}_{\mathrm{ij}} \cdot \mathbf{n}_{\mathrm{ij}} \mathrm{A}_{\mathrm{ij}} \tag{3.13}$$

where $nf(i)$ is the number of faces of the i^{th} control volume. For a hexahedral cell nf is six and for a triangular prismatic cell nf is five. The term $\mathbf{f}_{\mathrm{ij}}$ represents the numerical flux used to approximate viscous and convective fluxes, $\mathbf{F}$. Once these approximations in space have been made, Eq. 3.1 can be rewrtten as

$$\frac{\partial \mathbf{q}_{\mathrm{i}}}{\partial t} = \frac{1}{\Omega_i} \left(\mathbf{g}_{\mathrm{i}} \Omega_{\mathrm{i}} - \sum_{\mathrm{j}=1}^{\mathrm{nf(i)}} \mathbf{f}_{\mathrm{ij}} \cdot \mathbf{n}_{\mathrm{ij}} \mathrm{A}_{\mathrm{ij}} \right). \tag{3.14}$$

Eq. (3.14) assumes that mesh deformation does not occur. Temporal changes of cell volumes and the movement of control volumes are to be taken into account for deforming grids.

B. Convective Flux Calculation

As seen by Eq. (3.14), numerical computations are highly dependent on flux calculations. In high Reynolds number flows, viscous fluxes are localized near walls and surfaces. The rest of the flow is dominated by the convective fluxes. The Lower-Upper Symmetric Gauss-Seidel method, as shown later, will depend solely on convective flux calculations. Therefore, accurate solutions for the convective fluxes are crucial in the numerical solution to the Navier-Stokes Equations.

Roe's approximate Riemann solver is implemented to calculate convective fluxes. Flux computations are required repeatedly during numerical simulation of fluid flow. The Riemann problem is nonlinear and uses a great deal of computational effort.

Roe's approximation method linearizes the problem. Less computational cost is required with the technique and has a great deal of accuracy. The technique is desirable because of the low numerical dissipation it produces which allows for highly accurate results in boundary layers and near shocks.

The Riemann solver was originally formulated for a 1-D problem [2]. Roe's approximate Riemann Solver is extended to multi-dimensional problems using the grid aligned method. This technique rotates local coordinates along connecting interfaces between two cells and solves the 1-D problem.

The convective flux computed by Roe's approximate Riemann solver [2] is calculated between neighboring cell states $\mathbf{q}_\mathrm{i}$ and $\mathbf{q}_\mathrm{j}$.

$$\mathbf{f}_\mathrm{c} = \frac{1}{2}\left(\mathbf{F}_\mathrm{c}(\mathbf{q}_\mathrm{i}) + \mathbf{F}_\mathrm{c}(\mathbf{q}_\mathrm{j}) - |\hat{\mathbf{A}}_{roe}|(\mathbf{q}_\mathrm{j} - \mathbf{q}_\mathrm{i})\right) \tag{3.15}$$

Here $|\hat{\mathbf{A}}_{roe}|$ is the flux Jacobian computed with the cell averaged, conservative variables.

$$|\hat{\mathbf{A}}_{roe}| = \frac{\partial \mathbf{F}_\mathrm{ij}}{\partial \mathbf{Q}_\mathrm{i}} \tag{3.16}$$

The implementation of cell averaged values is the cause of the linearization to Riemann's problem. The solution to the convective flux is based on Roe's density weighted averages [3, 4].

$$\begin{aligned} \hat{\rho} &= \sqrt{\rho_i \rho_j} \\ \hat{u}_k &= \left(u_{ki}\sqrt{\rho_i} + u_{kj}\sqrt{\rho_j}\right) \,/\, \left(\sqrt{\rho_i} + \sqrt{\rho_j}\right) \\ \hat{H} &= = \left(H_i\sqrt{\rho_i} + H_j\sqrt{\rho_j}\right) \,/\, \left(\sqrt{\rho_i} + \sqrt{\rho_j}\right) \\ \hat{V} &= \hat{u}_k n_k - V_g \end{aligned} \tag{3.17}$$

The $|\hat{\mathbf{A}}|\Delta\mathrm{q}$ vector is pre-multiplied with dissipative terms as a vector sum. The Δ term represents the change between the j^{th} and i^{th} states. This pre-multiplication

allows greater computational efficiency.

$$
\begin{aligned}
|\hat{\mathbf{A}}|\Delta \mathrm{q} = |\hat{\mathrm{V}}| \left(\Delta\rho - \tfrac{\Delta \mathrm{p}}{\hat{\mathrm{c}}^2}\right) & \begin{bmatrix} 1 \\ \hat{u}_k \\ \frac{\hat{u}_k \hat{u}_k}{2} \end{bmatrix} \\
+|\hat{V}|\hat{\rho} & \begin{bmatrix} 0 \\ \Delta u_k - \Delta \hat{V} n_k \\ \hat{u}_k \Delta u_k - hatV \Delta \hat{V} \end{bmatrix} \\
+|\hat{V} - \hat{c}| \left(\tfrac{\Delta p - \hat{\rho}\hat{c}\Delta \hat{V}}{2\hat{c}^2}\right) & \begin{bmatrix} 1 \\ \hat{u}_k - \hat{c} n_k \\ \hat{H} - \hat{c}\hat{V} \end{bmatrix} \\
+|\hat{V} + \hat{c}| \left(\tfrac{\Delta p + \hat{\rho}\hat{c}\Delta \hat{V}}{2\hat{c}^2}\right) & \begin{bmatrix} 1 \\ \hat{u}_k + \hat{c} n_k \\ \hat{H} + \hat{c}\hat{V} \end{bmatrix}
\end{aligned}
\tag{3.18}
$$

Instabilities occur when the eigenvalue of the flux Jacobian approximation approach zero. Harten's entropy fix corrects the eigenvalue such that it never becomes zero [4]. The entropy fix also removes the artificial expansion shock caused by Roe's approximate solver with added numerical dissipation.

C. Time Integration

The following section describes the methods of time integration used by the flow solver. The *semi-discrete* equations are discretized temporally for the specific explicit and implicit solution methods. The Runge-Kutta explicit and Lower-Upper Symmetric Gauss-Seidel implicit time integration techniques are presented.

It is convenient to express the right hand side of Eq. (3.14) as a single structure,

$\mathbf{R}_\mathrm{i}$.

$$\mathbf{R}_\mathrm{i} = \mathbf{g}_\mathrm{i}\Omega_\mathrm{i} - \sum_{\mathrm{j}=1}^{\mathrm{nf(i)}} \mathbf{f}_\mathrm{ij} \cdot \mathbf{n}_\mathrm{ij}\mathrm{A}_\mathrm{ij}. \tag{3.19}$$

The time step calculated for each forward march into time is dependent on the CFL condition. The time step [5], Δt_i, is defined as

$$\Delta t_i = \sigma \frac{\Omega_i}{(\Lambda_i^x + \Lambda_i^y + \Lambda_i^z) + C(\tilde{\Lambda}_i^x + \tilde{\Lambda}_i^y + \tilde{\Lambda}_i^z)} \tag{3.20}$$

where σ is the CFL number and C is set to the value of 4. The convective spectral radii, Λ_i^j, are computed for each cell i for each direction j and are defined as

$$\Lambda_i^j = (|u_i| + c_i)\Delta A^j \tag{3.21}$$

where c_i is the speed of sound for the i^{th} control volume. The ΔA^j term is the projection of the cell volume Ω_i in the j^{th} direction.

$$\Delta A^j = \frac{1}{2} \sum_{k=1}^{nn(j)} |\Delta A_k n_j| \tag{3.22}$$

Where $nn(j)$ is the number of neighboring nodes to cell j in the first order stencil. The viscous spectral radii, $\tilde{\Lambda}_i^j$, are defined for each cell i in the jth direction.

$$\tilde{\Lambda}_i^j = max(\frac{4}{3\rho}, \frac{\gamma}{\rho})(\frac{\mu_L}{Pr_L} + \frac{\mu_T}{Pr_T})\frac{(\Delta A^j)^2}{V_i} \tag{3.23}$$

The governing equations are computed for continuous time steps. A solution to the flow is acceptable when the residual becomes insignificant. A steady flow condition does not rely on time accuracy. Thus calculations performed at differing cells progress through time at individual Δt_i. For an unsteady flow, time accuracy is required. A global time step is used for each cell and is the minimum Δt_i within the domain.

1. Explicit Time Integration

The method for explicit time integration utilizes a multi-stage Runge-Kutta integration scheme. The *semi-discrete* equations are integrated in time and the state variables are advanced to the next time step.

Substituting Eq. 3.19 into Eq. 3.14 gives

$$\frac{\partial \mathbf{q}_\mathrm{i}}{\partial t} = \mathbf{R}_\mathrm{i} \tag{3.24}$$

Using a forward difference scheme in time, Eq. (3.24) can be discretized temporally and is written as

$$\mathbf{q}_\mathrm{i}^{\mathrm{n}+1} = \mathbf{q}_\mathrm{i}^{\mathrm{n}} + \frac{\mathbf{R}_\mathrm{i}^{\mathrm{n}} \Delta \mathrm{t}_\mathrm{i}}{\Omega_\mathrm{i}} \tag{3.25}$$

where the current time step is represented with n and the future time step as $n+1$.

A four stage Runge-Kutta method is implemented in the code [6]. The state vector is updated using this four stage method by

$$\begin{aligned}
\mathbf{q}_\mathrm{i}^0 &= \mathbf{q}_\mathrm{i}^{\mathrm{n}} \\
\mathbf{q}_\mathrm{i}^1 &= \mathbf{q}_\mathrm{i}^0 + \alpha_1 \tfrac{\Delta \mathrm{t}_\mathrm{i}}{\Omega_\mathrm{i}} \mathbf{R}_\mathrm{i}(\mathbf{q}_\mathrm{i}^0) \\
\mathbf{q}_\mathrm{i}^2 &= \mathbf{q}_\mathrm{i}^0 + \alpha_2 \tfrac{\Delta \mathrm{t}_\mathrm{i}}{\Omega_\mathrm{i}} \mathbf{R}_\mathrm{i}(\mathbf{q}_\mathrm{i}^1) \\
\mathbf{q}_\mathrm{i}^3 &= \mathbf{q}_\mathrm{i}^0 + \alpha_3 \tfrac{\Delta \mathrm{t}_\mathrm{i}}{\Omega_\mathrm{i}} \mathbf{R}_\mathrm{i}(\mathbf{q}_\mathrm{i}^2) \\
\mathbf{q}_\mathrm{i}^4 &= \mathbf{q}_\mathrm{i}^0 + \alpha_4 \tfrac{\Delta \mathrm{t}_\mathrm{i}}{\Omega_\mathrm{i}} \mathbf{R}_\mathrm{i}(\mathbf{q}_\mathrm{i}^3) \\
\mathbf{q}_\mathrm{i}^{\mathrm{n}+1} &= \mathbf{q}_\mathrm{i}^4
\end{aligned} \tag{3.26}$$

where the stage coefficients [7], α_i, are

$$\begin{aligned}
\alpha_1 &= 0.1668 \\
\alpha_2 &= 0.3028 \\
\alpha_3 &= 0.5276 \\
\alpha_4 &= 1.0
\end{aligned} \tag{3.27}$$

2. Lower-Upper Symmetric Gauss-Seidel Implicit Scheme

In this section, the governing equations are reformulated in an implicit manner. The Lower-Upper Symmetric Gauss-Seidel implicit solution scheme is implemented to solve the system of equations [8, 9].

Applying a backward difference scheme for time integration to Eq. (3.14),

$$\frac{\Omega_i}{\Delta t_i}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n+1}} - \mathbf{q}_{\mathrm{i}}^{\mathrm{n}}) + \sum_{\mathrm{j=1}}^{\mathrm{nf(i)}} \mathbf{f}_{\mathrm{ij}}^{\mathrm{n+1}} \mathrm{A_{ij}} = 0 \tag{3.28}$$

where the assumption that there are no sources due rotation is enforced. A_{ij} is the area vector projected from node i to node j. The forward differences in time for q_i and $\mathbf{f}_{\mathrm{ij}}$ are defined as

$$\Delta \mathbf{q}_{\mathrm{i}}^{\mathrm{n}} = \mathbf{q}_{\mathrm{i}}^{\mathrm{n+1}} - \mathbf{q}_{\mathrm{i}}^{\mathrm{n}} \tag{3.29}$$

$$\Delta \mathbf{f}_{\mathrm{ij}}^{\mathrm{n}} = \mathbf{f}_{\mathrm{ij}}^{\mathrm{n+1}} - \mathbf{f}_{\mathrm{ij}}^{\mathrm{n}}. \tag{3.30}$$

Eq. (3.28) can be rewritten as

$$\frac{\Omega_i}{\Delta t_i}\Delta \mathbf{q}_{\mathrm{i}}^{\mathrm{n}} + \sum_{\mathrm{j=1}}^{\mathrm{nf(i)}} \Delta \mathbf{f}_{\mathrm{ij}}^{\mathrm{n}} \mathrm{A_{ij}} = -\sum_{\mathrm{j=1}}^{\mathrm{nf(i)}} \mathbf{f}_{\mathrm{ij}}^{\mathrm{n}} \mathrm{A_{ij}} \tag{3.31}$$

Neglecting rotational sources, the structure $\mathbf{R}$ has a different definition from that of Eq. 3.19.

$$\mathbf{R}_{\mathrm{i}}^{\mathrm{n}} = -\sum_{\mathrm{j=1}}^{\mathrm{nf(i)}} \mathbf{f}_{\mathrm{ij}}^{\mathrm{n}} \mathrm{A_{ij}} \tag{3.32}$$

Substituting Eqs. 3.32 into Eq. 3.31,

$$\frac{\Omega_i}{\Delta t_i}\Delta \mathbf{q}_{\mathrm{i}}^{\mathrm{n}} + \sum_{\mathrm{j=1}}^{\mathrm{nf(i)}} \Delta \mathbf{f}_{\mathrm{ij}}^{\mathrm{n}} \mathrm{A_{ij}} = \mathbf{R}_{\mathbf{i}}^{\mathbf{n}}. \tag{3.33}$$

It is useful to write the forward difference of the flux vector, $\mathbf{f}$, as

$$\Delta \mathbf{f}_{\mathrm{ij}}^{\mathrm{n}} = \left[\mathbf{f}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n+1}}, \mathbf{q}_{\mathrm{j}}^{\mathrm{n+1}}) - \mathbf{f}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n}}, \mathbf{q}_{\mathrm{j}}^{\mathrm{n+1}})\right] + \left[\mathbf{f}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n}}, \mathbf{q}_{\mathrm{j}}^{\mathrm{n+1}}) - \mathbf{f}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n}}, \mathbf{q}_{\mathrm{j}}^{\mathrm{n}})\right]. \tag{3.34}$$

The first term in Eq. 3.34 can be linearized [10].

$$\mathbf{f}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n+1}}, \mathbf{q}_{\mathrm{j}}^{\mathrm{n+1}}) - \mathbf{f}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n}}, \mathbf{q}_{\mathrm{j}}^{\mathrm{n+1}}) \approx \frac{\partial \mathbf{f}_{\mathrm{ij}}}{\partial \mathbf{q}_{\mathrm{i}}} \Delta \mathbf{q}_{\mathrm{i}}^{\mathrm{n}} \tag{3.35}$$

By defining the implicit operator, $\mathbf{D}$, as

$$\mathbf{D}_{\mathrm{i}} = \frac{\Omega_{\mathrm{i}}}{\Delta \mathrm{t}_{\mathrm{i}}} \mathbf{I} + \sum_{\mathrm{j}=1}^{\mathrm{nf(i)}} \frac{\partial \mathbf{f}_{\mathrm{ij}}}{\partial \mathbf{q}_{\mathrm{i}}} \mathrm{A}_{\mathrm{ij}}, \tag{3.36}$$

Eq. 3.33 can be rewritten.

$$\mathbf{D}_{\mathrm{i}} \Delta \mathbf{q}_{\mathrm{i}}^{\mathrm{n}} + \sum_{\mathrm{j}=1}^{\mathrm{nf(i)}} \left[\mathbf{f}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n}}, \mathbf{q}_{\mathrm{j}}^{\mathrm{n}} + \Delta \mathbf{q}_{\mathrm{j}}^{\mathrm{n}}) - \mathbf{f}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n}}, \mathbf{q}_{\mathrm{j}}^{\mathrm{n}}) \right] \mathrm{A}_{\mathrm{ij}} = \mathbf{R}_{\mathrm{i}}^{\mathrm{n}} \tag{3.37}$$

Similiar to Eq (3.15), the flux vector, $\mathbf{f}_{\mathrm{ij}}$, is approximated using the convective flux and the spectral radii [5].

$$\mathbf{f_{ij}} = \frac{\mathbf{1}}{\mathbf{2}} \left[\mathbf{f_i^c} + \mathbf{f_j^c} - \hat{\Lambda}_{\mathrm{i}}^{\mathrm{j}} \left(\mathbf{q_j} - \mathbf{q_i} \right) \right] \tag{3.38}$$

This linear approximation substitutes the spectral radii,

$$\hat{\Lambda}_i^j = |\mathbf{V}_{\mathrm{i}} \cdot \mathbf{n}_{\mathrm{ij}}| + \mathrm{c}_{\mathrm{i}} + \frac{2(\mu + \mu_{\mathrm{t}})}{\rho |\mathbf{n}_{\mathrm{ij}} \cdot (\mathbf{r}_{\mathrm{j}} - \mathbf{r}_{\mathrm{i}})|}, \tag{3.39}$$

for the flux Jacobian[8]. This substitution is permissible because the spectral radius is the eigenvalue of the flux Jacobian. In Eq. (3.39), $\mathbf{V}_{\mathrm{i}}$, c_i, μ, μ_t, and ρ are the velocity vector, local speed of sound, viscosity, turbulent viscosity, and density at node i. The position vectors of nodes i and j are given by $\mathbf{r}_{\mathrm{i}}$ and $\mathbf{r}_{\mathrm{j}}$. The projection from node i to j is $\mathbf{n}_{\mathrm{ij}}$. The numerical inviscid flux, $\mathbf{f}^{\mathrm{c}}$ is computed using Roe's approximate Riemann solver. This first order approximation increases the computational efficiency compared to explicit methods because the implicit operator, $\mathbf{D}$, reduces to a diagonal matrix [11].

$$\mathbf{D}_{\mathrm{i}} = \left(\frac{\Omega_{\mathrm{i}}}{\Delta \mathrm{t}_{\mathrm{i}}} + \sum_{\mathrm{j}=1}^{\mathrm{nf(i)}} \hat{\Lambda}_{\mathrm{i}}^{\mathrm{j}} \mathrm{A}_{\mathrm{ij}} \right) \mathbf{I} \tag{3.40}$$

The final discrete equation reduces to

$$\mathbf{D}_{\mathrm{i}}\Delta\mathbf{q}_{\mathrm{i}}^{\mathrm{n}} + \frac{1}{2}\sum_{\mathrm{j}=1}^{\mathrm{nf(i)}}\left[\mathrm{f}^{\mathrm{c}}(\mathrm{q}_{\mathrm{i}}^{\mathrm{n}}, \mathrm{q}_{\mathrm{j}}^{\mathrm{n}} + \Delta\mathbf{q}_{\mathrm{j}}^{\mathrm{n}}) - \mathbf{f}^{\mathrm{c}}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n}}, \mathbf{q}_{\mathrm{j}}^{\mathrm{n}}) - \hat{\Lambda}_{\mathrm{i}}^{\mathrm{j}}\Delta\mathbf{q}_{\mathrm{j}}^{\mathrm{n}}\right]\mathrm{A}_{\mathrm{ij}} = \mathbf{R}_{\mathrm{i}}^{\mathrm{n}}. \tag{3.41}$$

All information is known in Eq. (3.41) except for $\Delta\mathbf{q}_{\mathrm{j}}^{\mathrm{n}}$. The LUSGS implicit method is used to linearize the system of equations. The linearization negates the need to invert a matrix or solve a matrix equation. The matrix formed by the system of equations is sparse. A forward sweep through the grid is performed only considering the elements below the diagonal of the matrix. Intermediate values of $\Delta\mathbf{q}_{\mathrm{i}}^{\mathrm{n}}$ are computed and used to update the state vectors. A similar backward sweep is then executed with the upper triangular elements of the matrix. A final $\Delta\mathbf{q}_{\mathrm{i}}^{\mathrm{n}}$ is used to update the state variables to the next time step. The notation of $L(i)$ and $U(i)$ represent the lower and upper neighboring nodes for the i^{th} control volume. $L(i)$ consists of the elements below the diagonal of the matrix related to node i. Likewise, $U(i)$ represents the elements above the diagonal of the matrix related to node i. The Gauss-Seidel iteration method is executed in the following procedure.

Forward Sweep

$$\mathbf{D}_{\mathrm{i}}\Delta\mathbf{q}_{\mathrm{i}}^{\mathrm{n}} = \mathbf{R}_{\mathrm{i}}^{\mathrm{n}} - \frac{1}{2}\sum_{\mathrm{j}=1}^{\mathrm{L(i)}}\left[\mathbf{f}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n}}, \mathbf{q}_{\mathrm{j}}^{\mathrm{n}} + \Delta\mathbf{q}_{\mathrm{j}}^{\mathrm{n}}) - \mathbf{f}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n}}, \mathbf{q}_{\mathrm{j}}^{\mathrm{n}}) - \hat{\Lambda}_{\mathrm{i}}^{\mathrm{j}}\Delta\mathbf{q}_{\mathrm{j}}^{\mathrm{n}}\right]\mathrm{A}_{\mathrm{ij}} \tag{3.42}$$

Backward Sweep

$$\Delta\mathbf{q}_{\mathrm{i}}^{\mathrm{n}} = \Delta\mathbf{q}_{\mathrm{i}}^{\mathrm{n}} - \frac{\mathbf{D}_{\mathrm{i}}^{-1}}{2}\sum_{\mathrm{j}=1}^{\mathrm{U(i)}}\left[\mathbf{f}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n}}, \mathbf{q}_{\mathrm{j}}^{\mathrm{n}} + \Delta\mathbf{q}_{\mathrm{j}}^{\mathrm{n}}) - \mathbf{f}(\mathbf{q}_{\mathrm{i}}^{\mathrm{n}}, \mathbf{q}_{\mathrm{j}}^{\mathrm{n}}) - \hat{\Lambda}_{\mathrm{i}}^{\mathrm{j}}\Delta\mathbf{q}_{\mathrm{j}}^{\mathrm{n}}\right]\mathrm{A}_{\mathrm{ij}} \tag{3.43}$$

D. Grid Re-Ordering

The re-ordering method of node numbers for any given mesh is discussed. Unstructured meshes allow for fewer nodes in a mesh. Fewer nodes in a mesh alleviate intense memory usage and high computational costs. Unstructured meshes are created with

an arbitrary numbering system. Though structured meshes produce more grid points, a natural ordering system of the grid points are produced. Implicit techniques, by nature, depend on coherent ordering of the nodes within a mesh.

Structured grid re-ordering occurs about hyperplanes throughout the mesh [12]. A hyperplane is defined as the the group of nodes whose indices i, j, k add to the same number. When solving the governing equations on a structured mesh, the solution is advanced from one hyperplane to the next. When sweeping forward through the mesh, a solution is computed on a hyperplane at point (i, j, k) using updated values from the previous hyperplane with points $(i-1, j, k)$, $(i, j-1, k)$, and $(i, j, k-1)$. A similar backward sweep solution for (i, j, k) involves $(i+1, j, k)$, $(i, j+1, k)$, and $(i, j, k+1)$. The hyperplane method gives a natural division between upper and lower regions connected to each grid point.

Unstructured meshes have no formal numbering system. The hyperplane method can still be applied to unstructured meshes in conjunction with a re-ordering algorithm [12, 13]. The unstructured mesh re-ordering begins by selecting a starting node either at random or purposefully. This starting node is assigned as the first hyperplane. The nodes connected to the starting node are then assigned to the next hyperplane and so on. If there are connected nodes within a hyperplane, an intermediate hyperplane is introduced so no two nodes of the same hyperplane are connected. Figure (2) is the first hyperplane assignment to an unstructured grid. Nodes contained in a gray region are on even numbered hyperplanes. The nodes inbetween are the odd hyperplanes. Since there are connected nodes in the hyperplanes, further assignment occurs. Figure (3) represents the final hyperplane ordering.

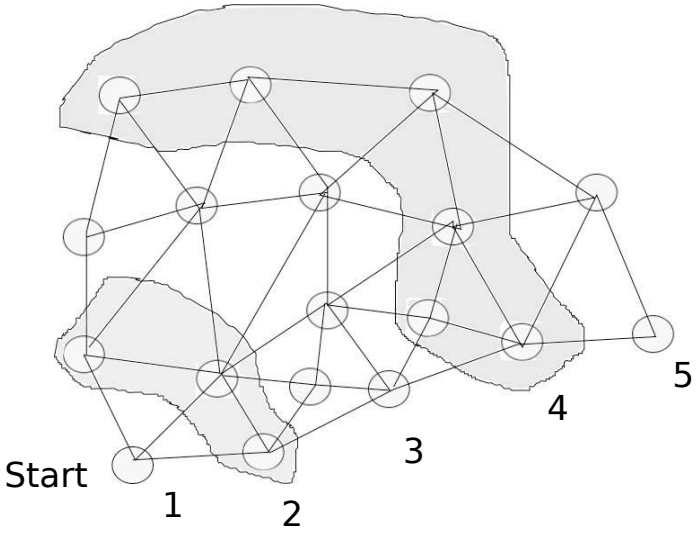

Fig. 2. First hyperplane iteratation.

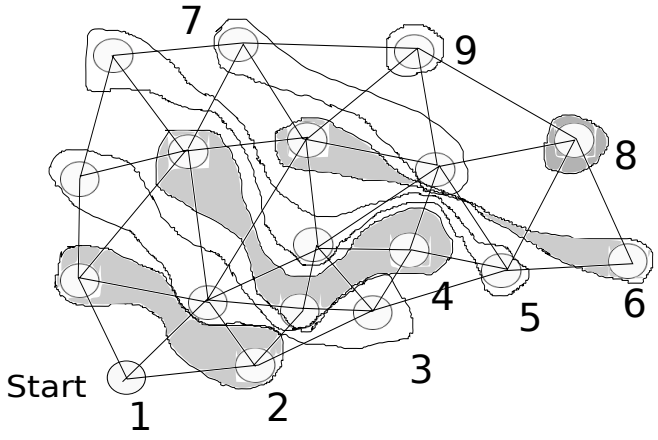

Fig. 3. Final hyperplane iteratation.

CHAPTER IV

NUMERICAL RESULTS

Several test cases were selected to validate the implemented LUSGS method. Tests were executed for incompressible, compressible, and transonic flow regimes. All cases were run as inviscid. External flows for NACA 0012 and 0015 airfoils were tested. The numerical simulations of the symmetric airfoils were run at two angles of attack, 0^o and 2^o. Also simulated was the transonic flow through a channel with a circular arc. The same geometries were run with the explicit code. Comparisons are made between the explicit and implicit results. For the flow through a channel, known data from literature is also compared. Finally, the Generic Transport Wing (GTW) is simulated for a compressible fluid. Results are compared between the explicit and implicit methods at the root and tip of the wing.

A. Inviscid Flow over NACA 0012 and 0015 Airfoils

The NACA 0012 and 0015 airfoils were tested for incompressible, compressible, and transonic flow regimes relating to Mach numbers of: 0.25, 0.6, and 0.85, respectively. The airfoils were tested at 0° and 2° angles of attack (AOA). A coarse and fine mesh were used to compute solutions for each airfoil at each AOA. The coarse meshes used 64 points to approximate the surface of the airfoils and 768 points are used in the structured mesh. Figures (4)-(5) present the coarse inviscid grids used for the numerical simulations. The finer meshes used 128 points to approximate the surface of the airfoils and 3200 points are used in the structured mesh. Figures (6)-(7) present the grids used for the numerical simulations of the finer NACA 0012 meshes. An entire view of each grid is given. A close up view of the structured mesh around the airfoils are also shown. Similar fine and coarse meshes were created using the NACA

0015 airfoil.

The CFL number was linearly ramped up to a large value. Once large enough, the CFL number has no effect on convergence rates. The CFL number was set to 0.5 for the explicit computations. The numerical simulations were executed first for the explicit cases. The explicit code would execute until residual convergence had occurred. The LUSGS code would then be run until the residual values had reached the converged values from the explicit results. Pressure coefficient, c_p, plots are compared for each case run. The c_p plots are given for a method to check the accuracy of the implicit scheme. Also given are the residual histories compared to the CPU time to study the convergence acceleration.

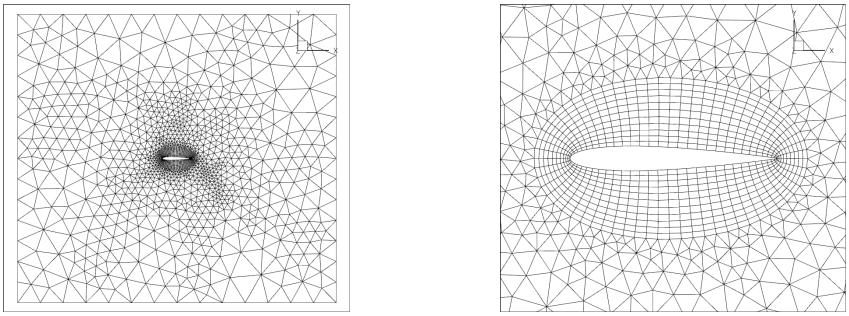

Fig. 4. NACA 0012 coarse mesh, 0° AOA.

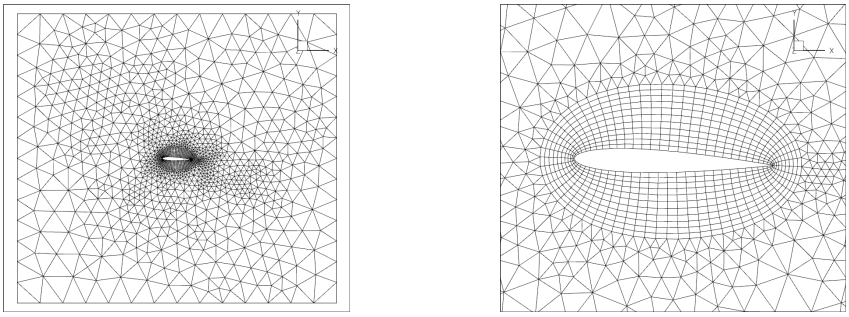

Fig. 5. NACA 0012 coarse mesh, 2° AOA.

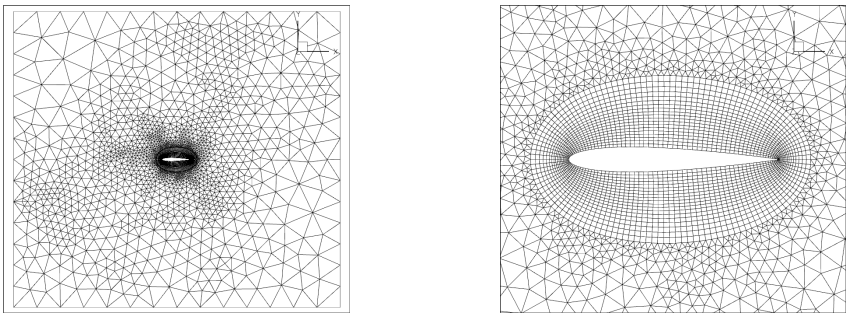

Fig. 6. NACA 0012 fine mesh, 0° AOA.

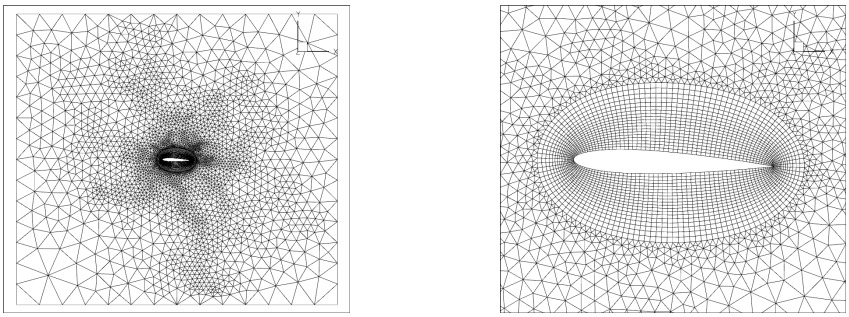

Fig. 7. NACA 0012 fine mesh, 2° AOA.

1. CFL Convergence Characteristics

Simultations were performed for a NACA 0012 airfoil at a Mach number of 0.85, with varying CFL numbers. The tests where conducted to study the effect of the CFL number on convergence characteristics. The grid used in the study was the fine mesh at 0° AOA, Figure (6).

The first test used a CFL number of 0.5 throughout the computations. This value was first employed because the explicit code uses the same value for the CFL number during execution. The CFL number was then initially set to the value of 1 and increased linearly over the first 100 iterations of the numerical experiments. The final values the CFL number was ramped up to were: 1, 10, 50, 100, 200, and 1000.

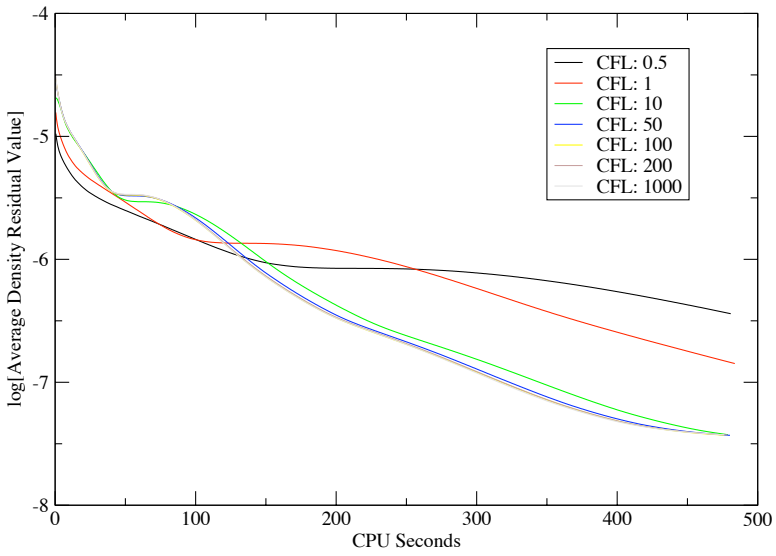

Fig. 8. Effect of CFL number on convergence rate.

Figure (8) presents the density convergence histories for the varying CFL number test cases. The figure reveals solutions obtained using LUSGS are indpendent of the CFL number greater than the value of 50.

2. Incompressible Test Cases

Simulations of the NACA 0012 and 0015 airfoils were computed using a freestream Mach number of 0.25. The implicit code would execute for 5000 iterations using spatial accuracy of first order. The CFL number was linearly ramped up from 1 to 200 for the first 100 iterations of each simulation. Then the implicit code would change from a first order to a second order accurate solution. The CFL number was then ramped up from 1 to 10 for 50 iterations. The explicit code would execute as a first order solutin for the first 500 iterations, then would execute as second order.

The NACA 0012 airfoils will be considered first. The results computed on the coarse meshes are shown in this study. Figures (9)-(10) present the c_p plots. The c_p plots demonstrate the accuracy of the LUSGS scheme by matching the solution

computed by the explicit solver. Discrepancies between the two methods are seen near the leading and trailing edges. The residual history plots are shown in Figures (11) and (12). The residual values shown in the figures are the L2 norm of the pressure values. It is seen from Figures (11) and (12) that LUSGS has a convergence speed of factor of about 2.5 when compared to the explicit solver for these grids.

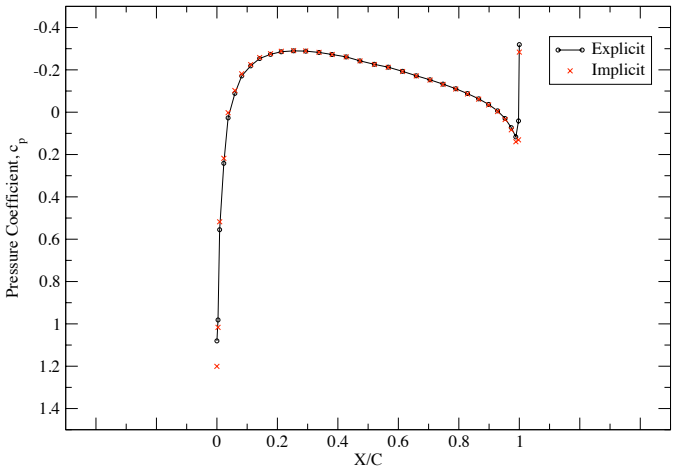

Fig. 9. Pressure coefficient, NACA 0012, 0° AOA.

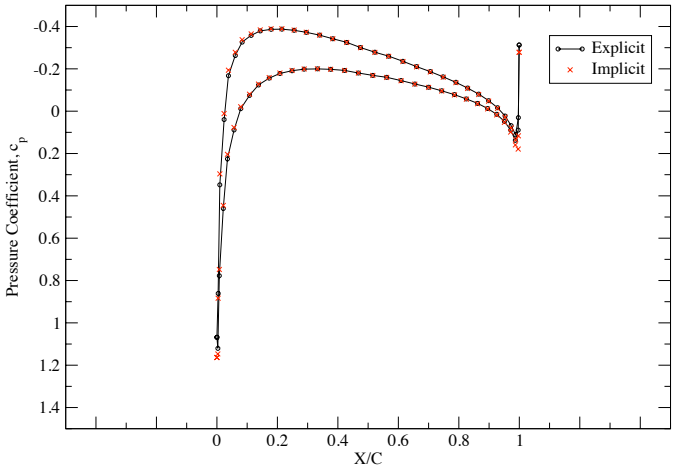

Fig. 10. Pressure coefficient, NACA 0012, 2° AOA.

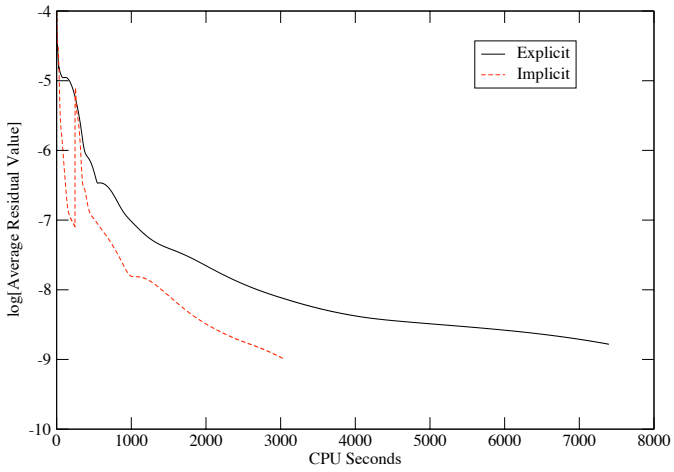

Fig. 11. Pressure residual history, NACA 0012, 0° AOA.

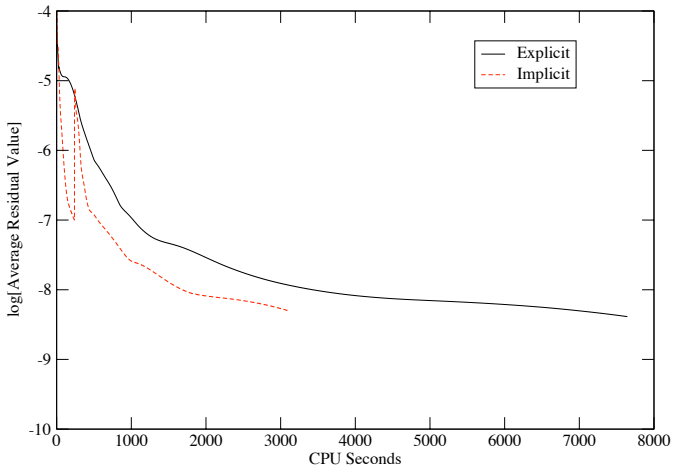

Fig. 12. Pressure residual history, NACA 0012, 2° AOA.

The NACA 0015 simulations reveal similar results computed to the NACA 0012 case. Figures (13) and (14) again show the high accuracy of LUSGS by matching the pressure coefficient calculated by the explicit code except near the leading and trailing edges. The speed up factor of 2.5 is still present in the pressure residual history plots, as seen in Figures (15) and (16).

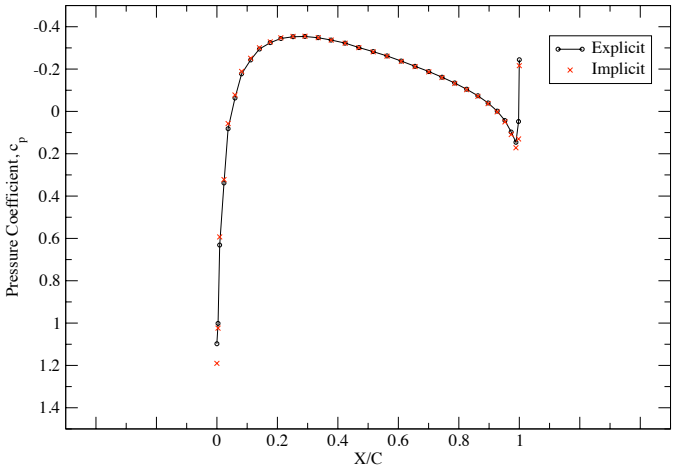

Fig. 13. Pressure coefficient, NACA 0015, 0° AOA.

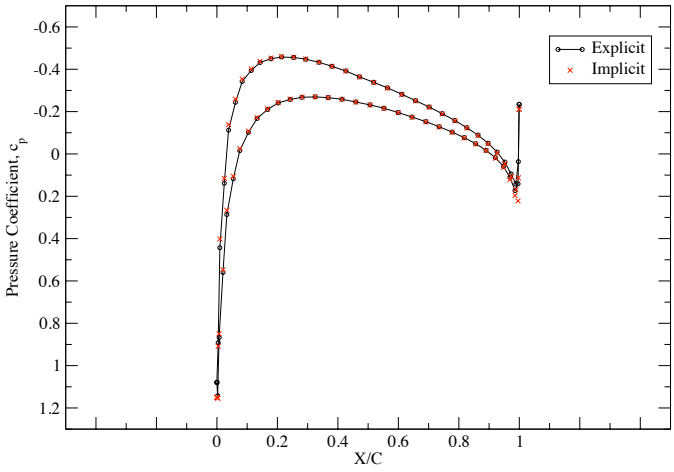

Fig. 14. Pressure coefficient, NACA 0015, 2° AOA.

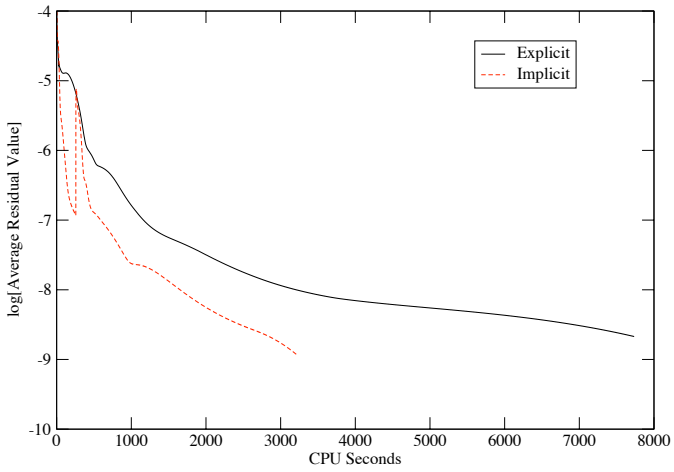

Fig. 15. Pressure residual history, NACA 0015, 0° AOA.

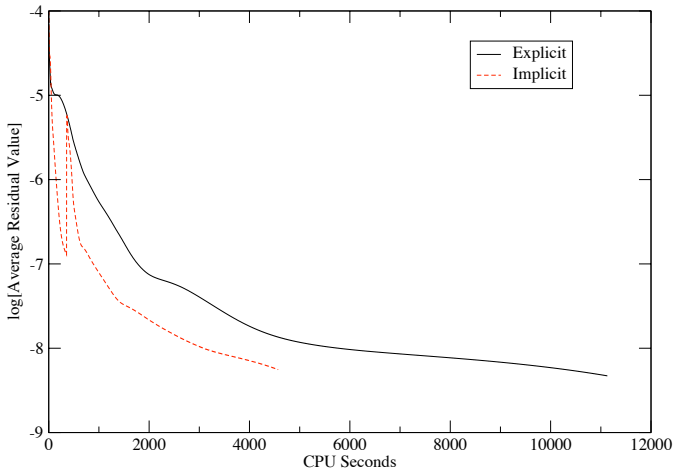

Fig. 16. Pressure residual history, NACA 0015, 2° AOA.

3. Compressible Test Cases

Simulations of the NACA 0012 and 0015 airfoils were computed using a freestream Mach number of 0.6. It is discernable from the previous section that the comparisons of the two airfoils are similar. From this section on the NACA 0012 airfoil case will only be considered. The results shown in this section were computed on the finer

meshes of the NACA 0012 airfoil, Figures (6) and (7).

Figures (17) and (18) present the pressure distributions along the chord of the airfoils. The leading and trailing edges are more aligned than with the previous studies involving the coarse meshes. There is, however, still slight discrepancy between the two solutions. The residual of the u velocities are shown in Figures (19) and (20). Although the u velocities are compared, rather than the pressure residuals from before, the speed up factor is still about 2.5.

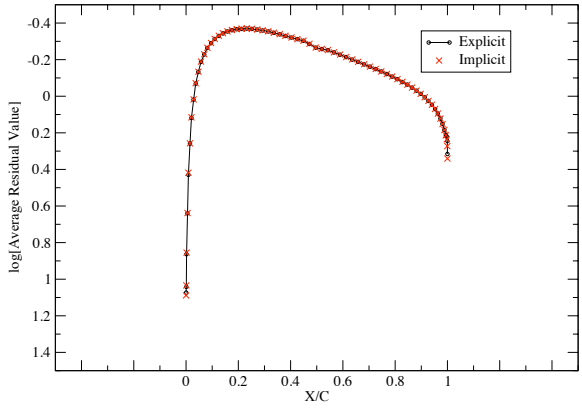

Fig. 17. Pressure coefficient, NACA 0012, 0° AOA.

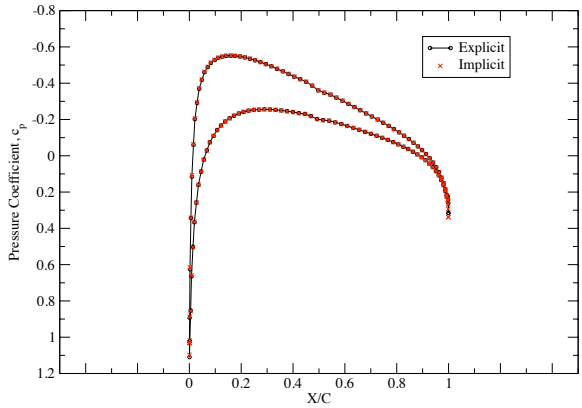

Fig. 18. Pressure coefficient, NACA 0012, 2° AOA.

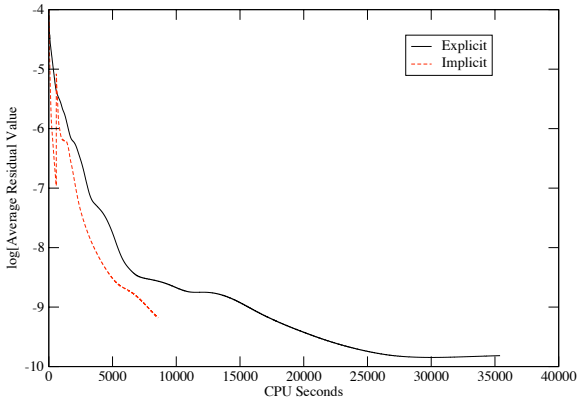

Fig. 19. U-velocity residual history, NACA 0012, 0° AOA.

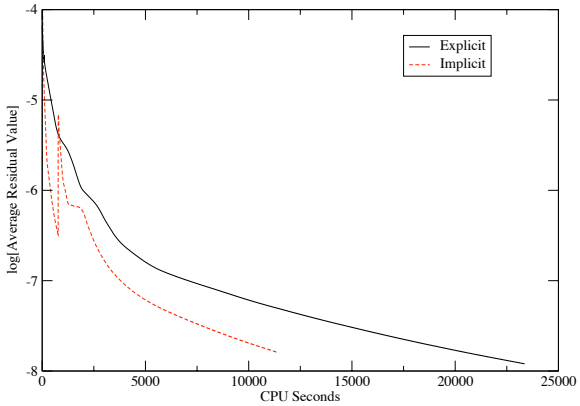

Fig. 20. U-velocity residual history, NACA 0012, 2° AOA.

4. Transonic Test Cases

Simulations of the NACA 0012 and 0015 airfoils were computed using a freestream Mach number of 0.85. The results computed using the NACA 0012 airfoils, Figures (6) and (7) are shown. The accuracy and convergence behaviors for the two airfoils are similar to the previous sections results. Figures (21) and (22) show the pressure distribution along the surface of the airfoil. The c_p distributions now lie completely on top of each other for this higher Mach number. Figures (23) and (24), show

the speed up factor for LUSGS to still be about 2.5. The speed up factor is seen as global because it has not changed for ranging Mach numbers for all variables: density, velocity, and pressure.

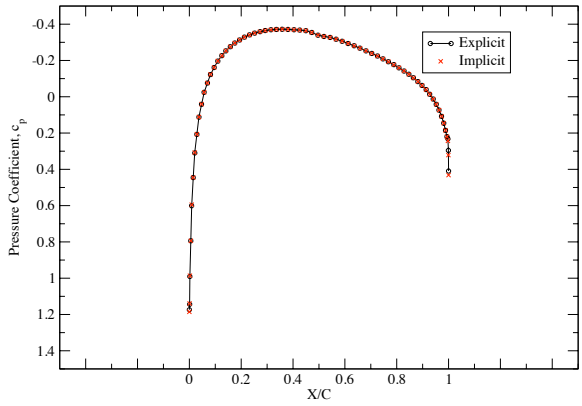

Fig. 21. Pressure coefficient, NACA 0012, 0° AOA.

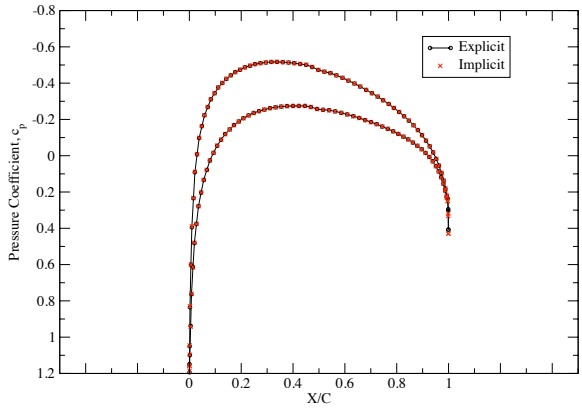

Fig. 22. Pressure coefficient, NACA 0012, 2° AOA.

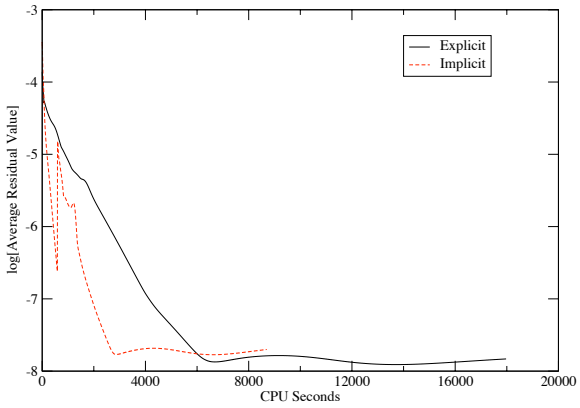

Fig. 23. Density residual history, NACA 0012, 0° AOA.

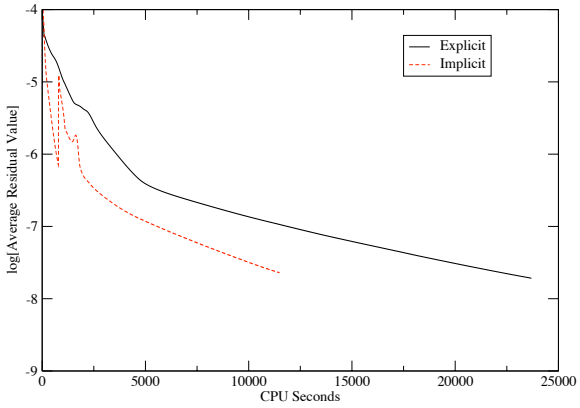

Fig. 24. Density residual history, NACA 0012, 2° AOA.

Sources [14] have shown shocks to form near the trailing edge of the NACA 0012 airfoil for this Mach number. A further study was conducted using even finer meshes for the NACA 0012 airfoil than from the previous experiments. Figures (25) and (26) show the refined structured meshes around the NACA 0012 airfoil used in this shock investigation.

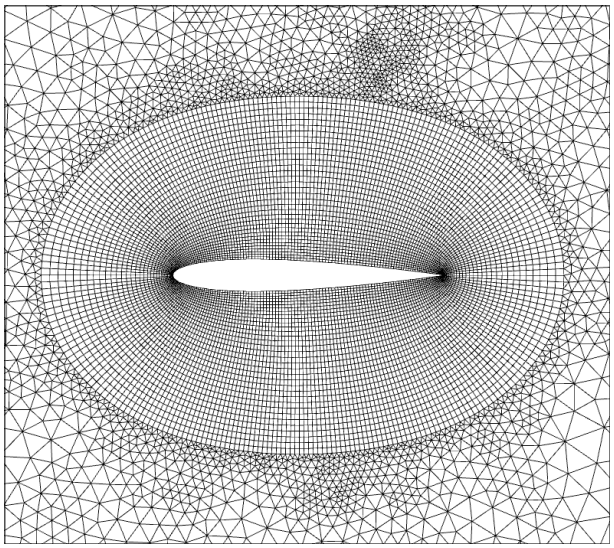

Fig. 25. NACA 0012 refined mesh, 0° AOA.

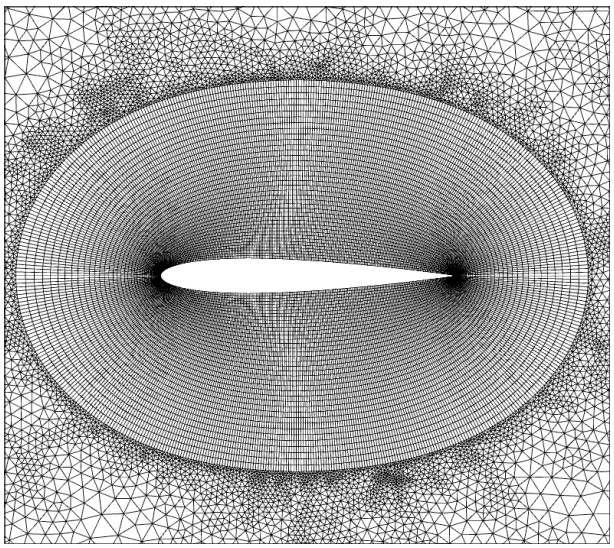

Fig. 26. NACA 0012 finest mesh, 0° AOA.

The pressure coefficient distributions are presented in Figures (27) and (28). The implicit and explicit pressure distributions now agree completely by having more points representing the airfoil. The shock begins to appear at 60% of the chord in Figure (27). The shock is smoothed out in the figure due to numerical dissipation brought on by the numerical scheme [4] and the spacing of grid points. The shock

near 60% chord location becomes more distinct as more points are used in the grid as seen in Figure (28).

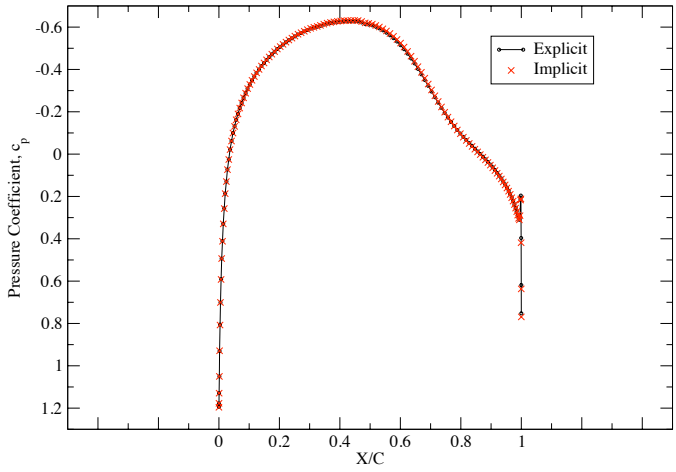

Fig. 27. Pressure coefficient, NACA 0012, 0° AOA, refined mesh.

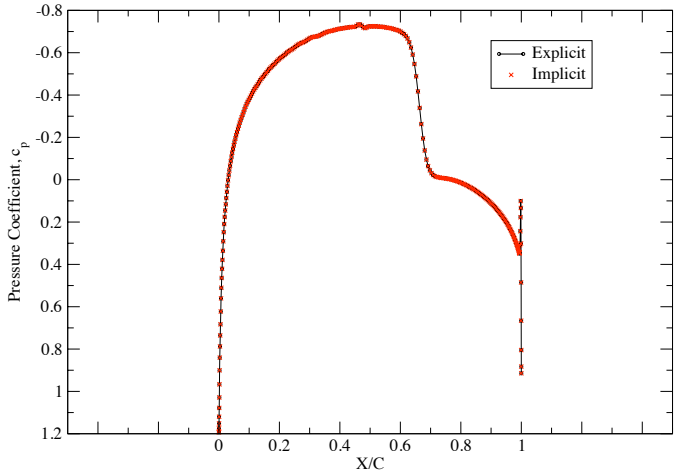

Fig. 28. Pressure coefficient, NACA 0012, 0° AOA, finest mesh.

The density residual plots for the refined mesh investigations are shown in Figures (29) and (30). As seen in the previous residual plots, the speed up factor between the explicit and implicit methods are around 2.5.

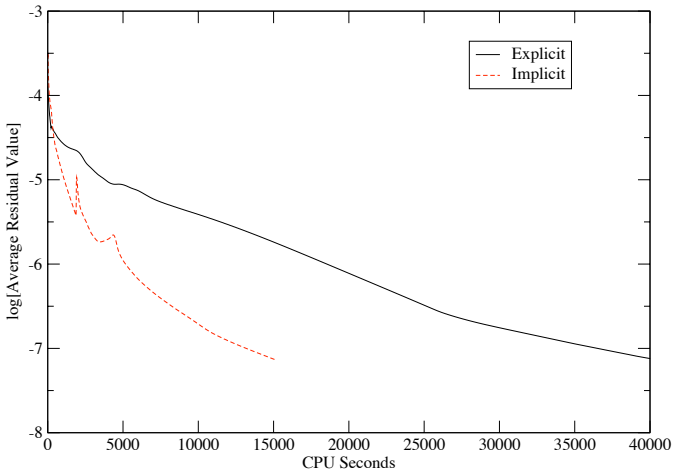

Fig. 29. Density residual history, NACA 0012, 0° AOA, refined mesh.

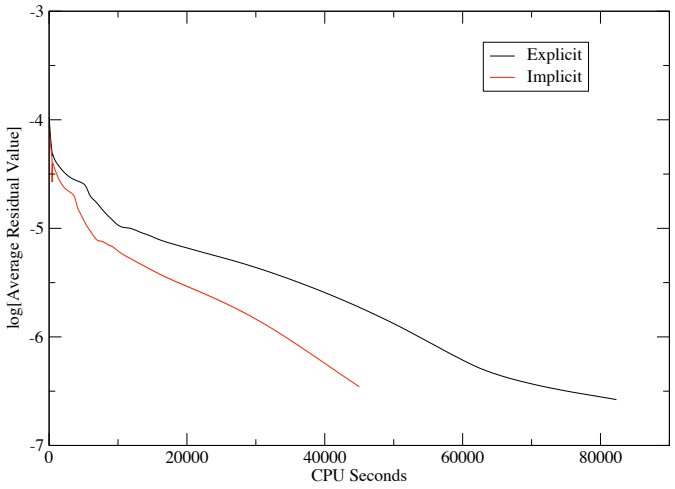

Fig. 30. Density residual history, NACA 0012, 0° AOA, finest mesh.

B. Inviscid, Transonic Flow Through a Channel with a Circular Arc

An internal geometry of a channel with a circular arc was tested at a transonic case and is shown in Figure (31). There are 72 grid points in the x-direction and 21 in the y-direction. There are 55 evenly distributed points in x-direction used in the region

surrounding the circular arc. The inlet Mach number for the channel was set to 0.85. The circular arc has a chord length of 1.0 and a height of 0.042. The height and length of the channel are 2.073 chord lengths tall and 5.0 chord lengths long. The leading edge of the arc is 2.0 chord lengths from the inlet. Figures (32) and (33) illustrate the Mach number in the channel for the LUSGS and explicit computations. The LUSGS solution is in great concordance with the explicit results, especially with shock location and shock height. The height of the shock for both cases is 0.67 chord lengths above the bottom surface. This agrees well with known solutions for the same geometry and flow regime [14]. Accepted solutions have shown a shock height of 0.62 to 0.67 chord lengths tall. The location of the shock on the bottom surface occurs at 84% of the chord of the arc. Results from other sources [14] have shown shock locations to occur at 77.5% to 85% of the chord on the arc. The density residual plot is shown in Figure (34). The speed up factor obtained in the NACA 0012 simulations is still prevalent for the flow through the channel test case.

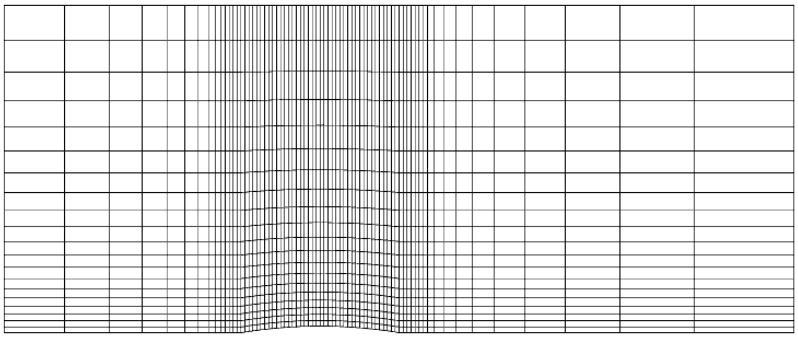

Fig. 31. Channel flow with circular arc geometry.

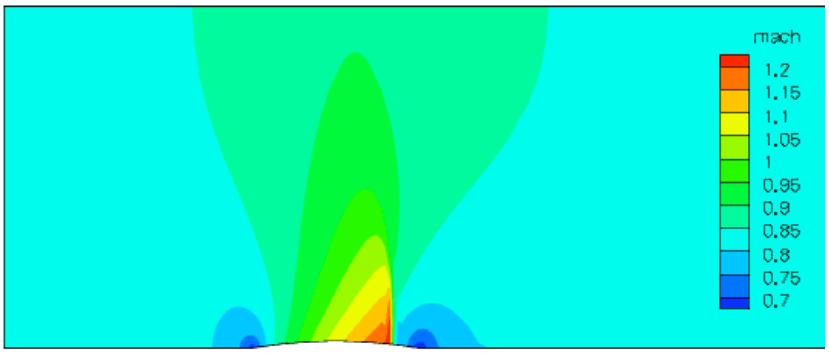

Fig. 32. LUSGS solution for channel flow with circular arc.

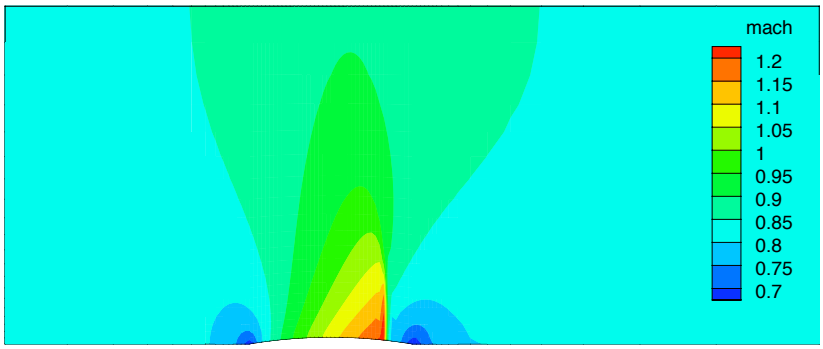

Fig. 33. Explicit solution for channel flow with circular arc.

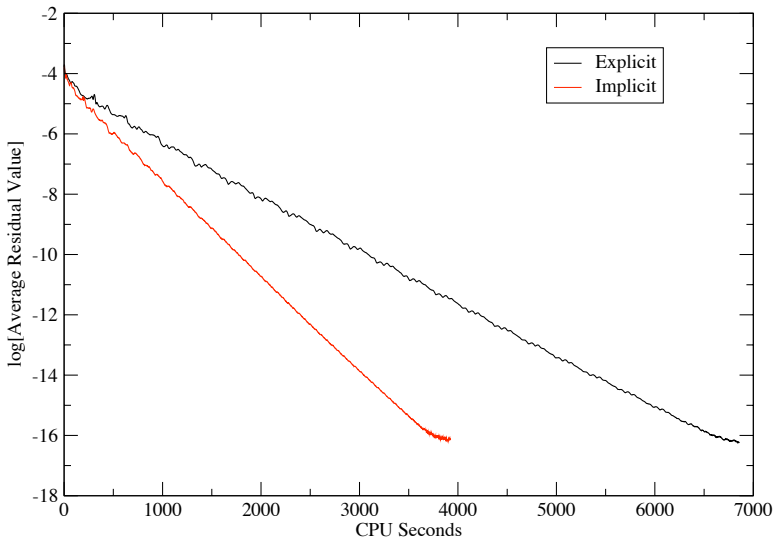

Fig. 34. Density residual history, channel flow with circular arc.

A comparison is made between previous numerical simulations and the LUSGS results. Figure (35) is numerical data obtained by Deconinck and Hirsch [14]. Figure (36) presents Mach contours of the same geometry. The figures show similar locations for the shock and similar flow characteristics. The current code predicts the shock to be slightly higher, 3% chord, and slightly further back, 3% chord, than literature.

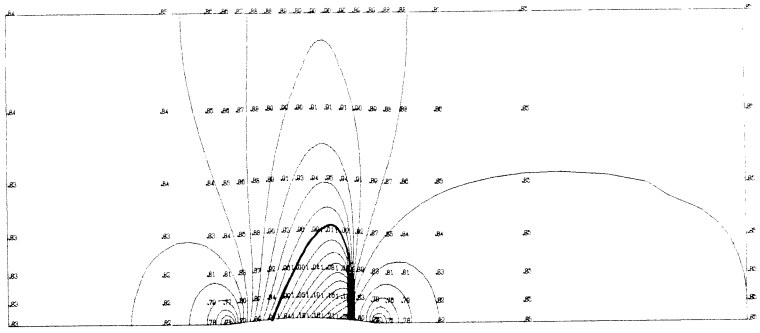

Fig. 35. Mach contours for channel flow with circular arc.

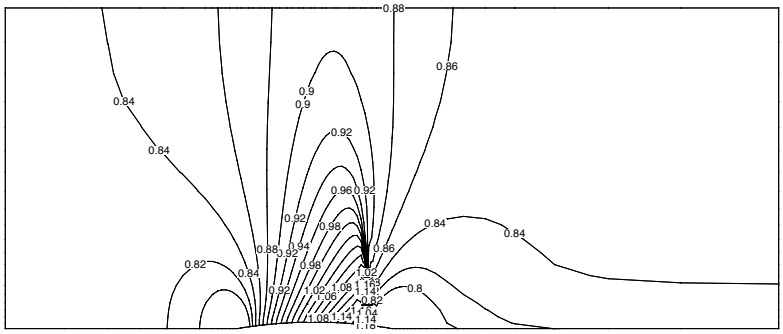

Fig. 36. LUSGS Mach contours for channel flow with circular arc.

C. Inviscid Flow over the Generic Transport Wing

The Generic Transport Wing was tested using the explicit and implicit solutions methods. The results obtained at the root and the tip locations are presented. The geometry of the GTW can be seen in Figure (37).

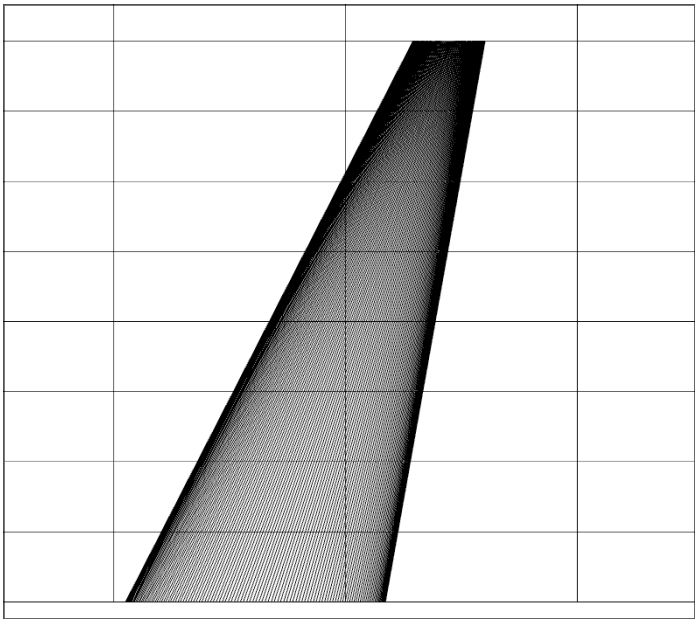

Fig. 37. Top view of the GTW.

The GTW was tested using a freestream Mach number of 0.6 at an AOA of 0°. Figures (38) and (39) show the pressure contours at the root of the GTW for both the explicit and implicit solutions methods. The contour plots reveal highly agreeable solutions between the two integrations schems by capturing the same flow characteristics.. The pressure contours at the tip of the GTW computed by the two schemes are presented in Figures (40) and (41). Similar to the solutions at the root, the pressure contours computed by the explicit code is in great agreement with the implicit solution.

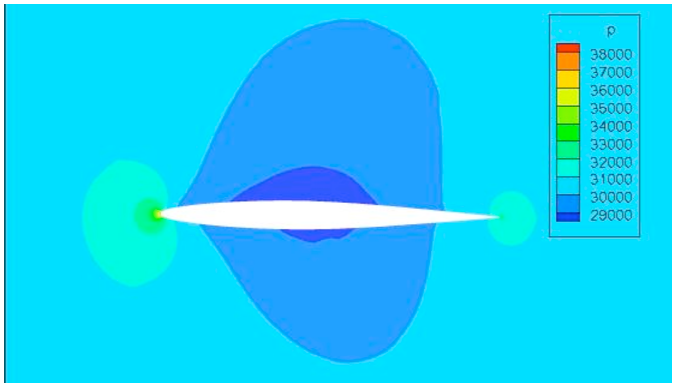

Fig. 38. Explicit solution of pressure at the root of the GTW.

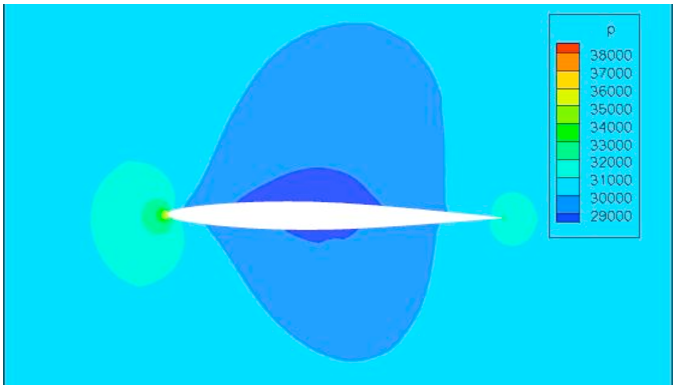

Fig. 39. LUSGS solution of pressure at the root of the GTW.

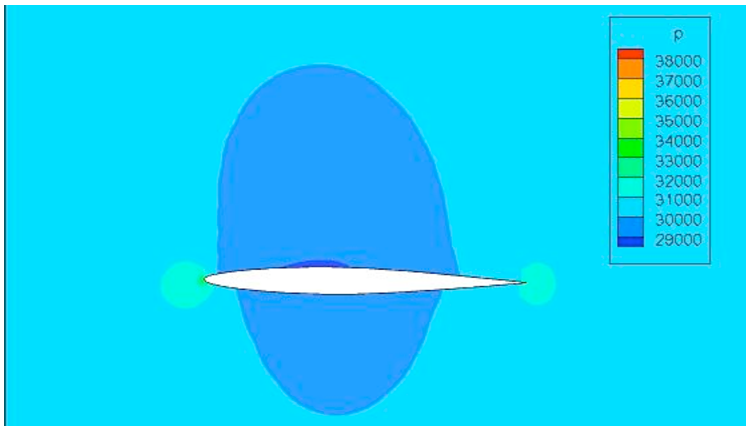

Fig. 40. Explicit solution of pressure at the tip of the GTW.

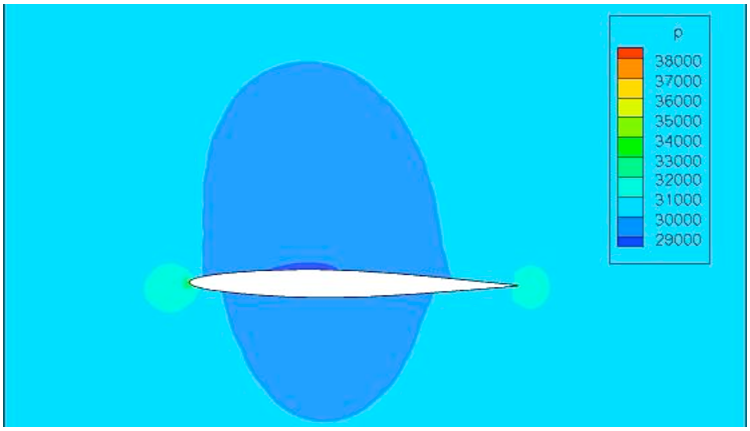

Fig. 41. LUSGS solution of pressure at the tip of the GTW.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

A. Conclusions

The flow solver was developed using the Finite Volume Method for unstructured grids using a median dual mesh technique. The governing equations were discretized over the control volumes of the mesh. The convective flux was calculated using Roe's approximate Riemann solver. The Runge-Kutta explicit time integration method was presented. The Lower-Upper Symmetric Gauss-Seidel implicit scheme was formulated.

The implicit scheme was validated using inviscid, symmetric airfoils at Mach numbers of 0.25, 0.6, and 0.85 for two angles of attack: 0° and 2°. Pressure distributions over the airfoils were compared. The results revealed agreement between the validated explicit solver and the implicit solver, especially as grids were refined. The results also showed as the Mach number increased, the two pressure distributions converged on top of each other. The residual histories were also compared. The implicit code revealed a universal 2.5 speed up factor for each state variable for each Mach number. Transonic channel flow with a circular arc was also simulated. The implicit method captured the same flow features computed by the explicit method with a similar speed up factor computed in the NACA 0012 simulations. The two methods also calculated the same shock locations within the channel and were compared to known computed solutions from literature. The shock locations were found to be within the ranges documented by previous authors. A final inviscid study was performed using the Generic Transport Wing at a Mach number of 0.6.. Previously computed results by the explicit method were compared to solutions obtained with LUSGS. The two

schemes are highly agreeable in the final calculated solutions.

B. Future Work

The discrepancy found in the transonic inviscid channel flow case will be studied further. Possible solutions could be found by allowing the codes to further converge. The future work will be to allow the code to run with viscous cases. Proper calculation of the viscous spectral radii are required. The final step will be to allow the implicit code to execute for unsteady flows. The addition of a disturbance time step will be requried in order to capture the unsteady effects while using arbitrarily large CFL numbers [15].

REFERENCES

[1] K. Kim and P.G. Cizmas, "Three Dimensional Hybrid Mesh Generation for Turbomachinery Airfoils," *Journal of Propulsion and Power*, vol. 18, no. 3, pp. 536-543, May 2002.

[2] P. Roe, "Approximate Riemann Solver, Parameter Vectors, and Difference Schemes," *Journal of Computational Physics*, vol. 18, 1986, pp. 337-365, Sep. 1988.

[3] M. Kermani, "Modified Entropy Correction Formula for the Roe Scheme," 39th AIAA Aerospace Sciences Meeting and Exhibit, AIAA Paper 2001-0083, Reno, NV, Jan. 2001.

[4] A. Harten, "Self Adjusting Grid Methods for One-Dimensional Hyperbolic Conservation Laws," *Journal of Computational Physics*, vol. 50, pp 235-269, 1983.

[5] J. Blazek, *Computational Fluid Dynamics: Principles and Applictatins*. New York: Elsevier 2001.

[6] C. Laney, *Computational Gasdynamics*. Press Syndicate of the University of Cambridge: Cambridge, United Kingdom, 1st edition, 1998.

[7] K. Kim, "Three Dimensional Hybrid Grid Generator and Unstructured Flow Solver for Compressors and Turbines," Ph.D. dissertation, Texas A&M University, College Station, Texas, Dec. 2003.

[8] R.F. Chen and Z.J. Wang, "Fast, Block Lower-Upper Symmetric Gauss-Seidel Scheme for Arbitrary Grids," *AIAA Journal*, vol. 38, no. 12, pp. 2238-2245, 2000.

[9] Z.J. Wang,"A Quadtree-Based Adaptive Cartesian/Quad Grid Flow Solver for the Navier-Stokes Equations," *Computers and Fluids*, vol. 27, no. 4, pp. 529-549, 1998.

[10] I. Men'shov I. and Y. Nakamura, "An Implicit Advection Upwind Splitting Scheme for Hypersonic Air Flows in Thermochemical Nonequilibrium," 6th Int. Symp. on CFD, Collection of Tech. Papers, 2 pp. 815-820, 1995.

[11] D. Sharov and K Nakahashi, "Low Speed Preconditioning and LU-SGS Scheme for 3-D Viscous Flow Computations on Unstructured Grids," AIAA Paper 98-0614, Jan. 1998.

[12] M. Soestrisno, S.T. Imlay, and D.W. Roberts, "A Zonal Implicit Procedure for Hybrid Structured-Unstructured Grids," AIAA Paper 94-0645, Jan. 1994.

[13] D. Sharov and K Nakahashi , "Reordering of 3D Hybrid Unstructured Grids for Vectorized LU-SGS Navier-Stokes Computations," AIAA Paper 97-2102, July 1997.

[14] A. Rizzi and H .Viviand, "Numerical Methods for the Computation of Inviscid Transonic Flows with Shock Waves," *Notes on Numerical Fluid Mechanics*. Friedr. Vieweg & Sohn: Wiesbaden, Braunschweig Germany, 1981.

[15] X. Zhao, S.J. Zhang, and A. Meganathan, "Implicit Time-Accurate Method for Unsteady Computations," *AIAA Paper* 2009-116.

APPENDIX A

SOURCE CODE: MAIN.F90

This is the source code of the main program to the in house code presented in this dissertation.

```
program uns3d

! Unstructured 3d flow solver
!
! modified by Kyusup Kim in 2002
! modified by Joaquin Ivan Gargoloff in 2004 - 2005
!
! Old NOTE: The viscous / turbulent and master / slave periodic
! aspects
! of the flow solver have not been addressed yet.

! jig - 2.0 - 08/02/04 - added Least-squares QR decomposition
!                        option
! jig - 3.0 - 02/03/06 - added laminar flow capabilities
! jig - 4.0 - 02/09/06 - added turbulent flow capabilities
! pgc - 4.4 - 12/22/07 - added stamp
! pgc - 4.5 - 07/09/08 - added debug in mod/flag.f90
! pgc - 4.6 - 07/15/08 - changed invis, lamin to logical
!             11/04/08 - added itersave (hardwired)

use bndr_vars,   only: qb
use constants,   only: zero
use cvari,       only: dtime, limiter_temporal, pinlet_leak, rramp &
     rramp_leak, time, omegax
use flag,        only: debug
use flow_vars,   only: q, q1, q2, res
use ibase,       only: igeom, invis, iorder, lamin, lsgg, mtime, &
                       steady
use imp
use implct
use io_unit,     only: res_io
use mesh_vars,   only: cv, nbface, ncell, nedge, nface, nnode, &
     np_bface, xnd, ynd, znd
use rscal,       only: scale, vref
use switch,      only: dump_tecplot, dump_yplus, intev_freq, &
     iramp, iramp_lim, iramp_lim0, iramp0, iramp0_leak, iturm, &
     leak_outlet, mstg, relative_v, res_display, res_freq, &
     reset_iter_counter
use turb_vars,   only: fbt, qt
use visc_vars,   only: fmu, tur
use work_arrays, only: rk

implicit none

!for precision
integer,parameter :: dbl = kind(0.0d0)

! local
real(8),allocatable::bf2(:)   ! turbulent dissipation coefficient
character(64) :: case_name    ! case name, used to generate the
!                               filenames for y+ info files
real(8) :: cfl                ! cfl number
real(8) :: cflr               ! cfl number for Runge-Kutta
real(8) :: dtrd               ! delta time
character(64) :: filedin      ! name of the flow field input file with
!                               the last flow field backup
character(64) :: filedout     ! name of the flow field backup, this
!                               file is used as the flow field input
!                               in the next run
character(64) :: fileturin    ! name of input restart file of
!                               turbulence variables
character(64) :: fileturout   ! name of output restart file of
!                               turbulence variables
!real(8) :: fmax               ! maximum value of residual  from V46
character(64) :: gridfile     ! name of the pre-processed grid data,
!                               this file is generated by the grid
```

```
!                               pre-processing code
integer :: i                   ! node index
integer :: ilnode_max          ! number of rotor / leak interface nodes
integer :: inode_rmax(5)       ! node id of the maximum residual
character(64) :: input         ! name of the input file
integer :: istd                ! to use unsteady init flow field! BUG b/c
!                               is always 0
integer :: istep               ! flag that tells the initialization of
!                               the flow field, start from a uniform
!                               field (0), or read the flow field from
!                               filein
integer :: istep1              ! istep1 = istep + 1
integer :: iter                ! pseudo time marching step
integer :: itime               ! real time marching step
logical :: lastiter            ! last computed iteration flag
integer :: nstep               ! total number of time marching steps
integer :: ntime               ! number of time marching steps
real(8) :: om                  ! = 1 if q is interpolated from q1 and
!                               q2, otherwise = 0 and q = q1
real(8) :: rmax(5)             ! maximum residual value
real(8) :: rms(5)              ! average residual value
real(8) :: rmaxt(2)            ! maximum residual value for turbulent model
real(8) :: rmst(2)             ! average residual value for turbulent model
! real(8) :: rmst                           ! from V46
character(64) :: tecplot_name ! filename for output in tecplot format
character(64) :: tempin       ! name of the unsteady flow field file
!                               (both in / out)
real(8) :: ttime               ! total time - unsteady flows
integer :: itersave            ! save solution every itersave iterations
integer :: status

character(32) :: impfile
integer :: cpuid, cpusv, cpulcv
real(kind=dbl) :: cpt1, cpt2, cpt, temp
!integer :: cfliter
real(kind=dbl) :: dcfl
! ----------------------------------------------------------------------
itersave = 100
! get the name of the input file
call parse_args(input)

! read input named-lists and initialize input/output files
call read_input(input, ncell, nnode, nedge, nface, nbface, istep, &
     ntime, cfl, istd, gridfile, filedin, filedout, fileturin, &
     fileturout, tempin, tecplot_name, case_name, ttime)

! allocate arrays
!call conf_arrays(mstg, iorder, lamin, mtime, invis, lsgg)
call alloc1_arrays(mstg, iorder, lamin, mtime, invis, lsgg)

! set the values of the Runge-Kutta coefficients
call rk_coef(mstg, cflr, iorder)

! nondimensionalization
call nondimension

! read in mesh
call readstruc(gridfile)

! Compute geometric data, directed surface areas and volumes
call geomcalcul

! for implicit residual smoothing, precalculate the weighting
call irs_weight

! calculate geometric variables of least-square reconstruction
if (lsgg .eq. 1) then
   call cls
else if (lsgg .eq. 2) then
   call clsqr ! using QR decomposition
end if

! initialize flowfields
call initialfield(istep, q, qb, filedin, ilnode_max)

! Populate structures with lower and upper region edge data
!******* need to implement impfile into input file ******!
impfile = "imp.def"
call read_impstruct(nnode, impfile)

if (.not.invis) then ! viscous flow
   ! calculate dynamic viscosity coefficients
   call mukin(nnode, q, fmu)

   if (.not.lamin) then ! turbulent flow
      ! initialize turbulence flowfields
      call initurb(nnode, iturm, qt, nbface, np_bface, fbt, fileturin)
```

```
      ! calculate turm
      allocate (bf2(nnode), stat = status)
      if (status /= 0) then
         write (*,*) 'failure to allocate memory for bf2 in main', nnode
         stop
      end if
      do i = 1, nnode
         bf2(i) = 0.0 ! initialize turbulent dissipation coefficient
      end do
      call komega(nnode, qt, q(:, 1), fmu, tur, bf2)

      deallocate (bf2, stat = status)
      if (status /= 0) stop 'failure to deallocate memory for bf2 in main'
   end if
end if

! mtime : number of real time steps to be taken for unsteady dual
! time step
if (mtime .gt. 1) then
   ! initialize unsteady fields q1, q2
   call initialtime(istd, nnode, q1, q2, q, cflr, dtrd, tempin)
end if

nstep  = istep + ntime
istep1 = istep + 1
rmst   = 0.0

! real-time marching
do itime = 1, mtime
   if (itime .le. 2 .and. istep1 .le. 1) then
      om = 0.0
   else
      om = 1.0
   end if

   ! initial solution for new real-time step (three-point backward
   ! interpolation)
   if (mtime .gt. 1) then
      call q012(6, nnode, q1, q2, q, om)
   end if

   ! integrate physical parameters
   iter = istep1
   rmax = 0.0
   inode_rmax = 1

   call intev_wrapper(leak_outlet, iter, rmax, inode_rmax)

   lastiter = .false.

   dcfl = (cfln-cfl)/real(cfliter)

   ! iter loop -----------------------------------------------------
   do iter = istep1, nstep ! pseudo-time marching

      ! gradual change of boundary condition via rramp
      if (mtime .eq. 1) then
         call ramp_bc(iramp0, iramp, rramp, leak_outlet, rramp_leak, &
              iramp0_leak, pinlet_leak, nnode, q, cv, ilnode_max, &
              iramp_lim0, iramp_lim, limiter_temporal, &
              reset_iter_counter, iter)
      end if

      ! calculate dynamic viscosity coefficients
      if (.not.invis) then ! viscous flow
         call mukin(nnode, q, fmu)
      end if

      if (mod(iter, res_freq) .eq. 0) then
         res_display = .true.
      else
         res_display = .false.
      end if

      ! calculate cpu time before time integration

      cpusv = iter/res_freq
      cpuid = 351

      if (iter .eq. istep1 .AND. .true.) then
         !open(unit=cpuid,file='cpumon.dat',action='write',position='append')
         open(unit=cpuid,file='cpumon.dat',position='rewind')
         if (cpusv .eq. 0) then
            cpt = 0.0
         else
            do cpulcv = 1, cpusv-1, 1
               read(cpuid,*)
```

```
            end do
            read(cpuid,*) cpt, temp
         end if
      end if

      call CPU_TIME(cpt1)

      ! perform the implicit LU-SGS calculations
      call lusgs(mstg, rk, rms, rmax, inode_rmax, res_display, &
           dtrd, cfl, ttime, lastiter, invis, lamin, iter, istep1, nstep, &
           ncell, fmu, tur, maxedge, edgnbr, lnbr, omegax, rramp, &
           zero,impord)

      ! calculate cpu time after time integration

      call CPU_TIME(cpt2)

      cpt = cpt + cpt2 - cpt1

!       write(*,*)dtime*scale/vref

      ! turbulent viscosity
      !if (.not.lamin) then     ! from V46
      if (.not. invis .AND. .not.lamin) then
         !call mutura(mstg, rk, rmst, fmax, fmu, tur)
         call mutura(mstg, rk, fmu, tur, res_display, lastiter, rmst, rmaxt)
      end if

      ! Averaged L2 residual
      if (res_display) then
         if (intev_freq .ne. 0) then
            if (mod(iter, intev_freq) .eq. 0) then
               call intev_wrapper(leak_outlet, iter, rmax, inode_rmax)
            end if
         end if

         ! write the average and maximum residual information
         !call write_residual(rms, rmst, lamin, itime, iter, &
         !     inode_rmax, fmax, res_io, rramp, rmax, istep1)  ! from V46

         !call write_residual(rms, rmst, invis, lamin, itime, iter, &
         !     inode_rmax, rmaxt, res_io, rramp, rmax, istep1)  ! for V47

         call write_residual(rms, rmst, invis, lamin, itime, iter, &
              inode_rmax, rmaxt, res_io, rramp, rmax, istep1, cpt, cpuid)
      end if

      if (iter .eq. nstep .AND. .true.) then
         endfile(cpuid)
         close(cpuid)
      end if

      ! if ttime has been reached, then exit the iteration loop
      if (lastiter) then
         nstep = iter    ! correct number of iterations
         exit
      end if

      if (steady .and. mod(iter, itersave) == 0) then
         ! backup state variables (without turbulent variables)
         call wrest(6, nstep, nnode, q, nbface, qb, filedout) !TODO: n(i)step

         ! backup turbulent variables
         !if (.not.lamin) then   ! from V46
         if (.not.invis .AND. .not.lamin) then
            call wturb(nnode, qt, nbface, fbt, fileturout)
         end if

         ! backup state variables at itime-1 (q1) and itime-2 (q2)
         !if (istd .gt. 0) then  !BUG - istd is always 0 and it should not be
         if (.not.steady .and. mtime > 1) then
            call backq12(6, nnode, q1, q2, time, tempin)
         end if
   end if

   if (iter-istep1 .le. cfliter) then
      cfl = cfl + dcfl
   end if

   end do
   !   iter loop ------------------------------------------------
   !   end of the pseudo-time marching

   if (itime == mtime .or. mod(itime, itersave) == 0) then
      ! backup state variables (without turbulent variables)
      call wrest(6, nstep, nnode, q, nbface, qb, filedout) !TODO: n(i)step
```

```
        ! backup turbulent variables
        !if (.not.lamin) then   ! from V46
        if (.not.invis .AND. .not.lamin) then
           call wturb(nnode, qt, nbface, fbt, fileturout)
        end if

        ! backup state variables at itime-1 (q1) and itime-2 (q2)
!        if (istd .gt. 0) then   !BUG - istd is always 0 and it should not be
        if (.not.steady .and. mtime > 1) then
           call backq12(6, nnode, q1, q2, time, tempin)
        end if
     end if
     time = time + dtime * scale / vref
  end do

  ! integrate physical parameters
  if (intev_freq .ne. 0 .and. .not.steady) then
     call intev_wrapper(leak_outlet, iter, rmax, inode_rmax)
  end if

  ! generate the tecplot output file
  if (dump_tecplot) then
     !call tecout(tecplot_name, 6, nnode, ncell, xnd, ynd, znd, q, res, &
     !     relative_v, igeom, lsgg, fmu, tur, lamin)  ! from V46

     call tecout(tecplot_name, 6, nnode, ncell, xnd, ynd, znd, q, res, &
          relative_v, igeom, lsgg, fmu, tur, invis, lamin)
  end if

  ! generate the y+ information for the viscous cases
  if (dump_yplus .AND. (.not. invis)) then
     call yplus(q, fmu, trim(case_name), lsgg)
  end if

  ! stamp
  call stamp(res_io)

  stop
end program uns3d
```

APPENDIX B

SOURCE CODE: LUSGS.F90

This is the source code of the implicit integration scheme to the in house code presented in this dissertation.

```
subroutine lusgs(mstg, rk, rms, rmax, inode_rmax, res_display, dtrd, &
     cfl, ttime, lastiter, invis, lamin, iter, istep1, nstep, ncell, fmu, tur, &
     maxedge, edgnbr, lnbr, omegax, rramp, zero, impord)

  ! explicit Runge-Kutta time integration

  ! jig - 09/01/04 - cleaned up
  ! pgc - 4.5 - 07/09/08 - added debug in mod/flag.f90
  ! pgc - 4.6 - 07/15/08 - changed invis to logical

  use bndr_vars, only: qb, sbs, sbx, sby, sbz
  use gasprop,   only: gm1, gamma, xgm1
  use flag,      only: debug
  use flow_vars, only: emax, emay, emaz,  q, q0, q1, q2, tx, ty, tz, res, &
       ux, uy, uz, vx, vy, vz, wx, wy, wz, px, py, pz
  use ibase,     only: mtime, igeom, iorder, lsgg, typlim
  use icntl,     only: irhsm
  use highvar,  only: e2, phi_lim
  use mesh_vars, only: cv, dtv, idbcs, ij_edge, ip_bface, nbface, nedge, &
       nnode, np_bface, ss, sx, sy, sz, xnd, ynd, znd, sxi, syi, szi
  use switch,    only: maxmachthreshold, monitormaxmach, relative_v
  use visc_vars, only: vres

  implicit none

  !for precision
  integer,parameter :: dbl = kind(0.0d0)
  ! input
  integer,intent(in) :: iter
  integer,intent(in) :: nstep
  integer,intent(in) :: istep1
  integer,intent(in) :: ncell
  integer,intent(in) :: maxedge
  integer,intent(in) :: edgnbr(nnode,maxedge)
  integer,intent(in) :: lnbr(nnode)
  real(8),intent(in) :: cfl           ! cfl number
  logical,intent(in) :: invis
  logical,intent(in) :: lamin
  real(8),intent(in) :: omegax
  real(8),intent(in) :: rramp
  real(8),intent(in) :: zero
  integer,intent(in) :: mstg          ! Runge-Kutta stages, (2, 3, 4 or
  !                                    5)
  logical,intent(in) ::res_display   ! if (T) compute the L2-norm of the
  !                                    residuals
  real(8),intent(in) ::ttime         ! total time - unsteady flows
  integer,intent(in) :: impord

  ! output

  integer,intent(out)::inode_rmax(5) ! node id of the maximum residual
  logical,intent(out)::lastiter      ! last computed iteration flag
  real(8),intent(out)::rmax(5)       ! maximum residual value
  real(8),intent(out)::rms(5)        ! average residual value
  real(8),intent(out)::fmu(nnode)
  real(8),intent(out)::tur(nnode)

  ! local

  integer :: neq
  integer :: status

  real(kind=dbl),allocatable :: lams(:)     ! spectral radius for each node
  real(kind=dbl),allocatable :: D(:)        ! D matrix for LU-SGS calculations
  real(kind=dbl),allocatable :: qp(:,:)     ! temporary matrix used for sweep
  real(kind=dbl),allocatable :: dq(:,:)
```

```
real(kind=dbl),allocatable :: qmin(:,:)
real(kind=dbl),allocatable :: qmax(:,:)
integer :: m          ! dummy variable - mstg
real(8) :: rk(mstg)   ! Runge-Kutta coefficients
real(8) :: mach_max   ! maximum mach number value
integer :: max_mach_i ! maximum mach number cell index
!integer :: i
real(8),intent(in) ::dtrd

! ---------------------------------------------------------------------
if (debug) write(*,*) 'Calling lusgs'

! q0(nnode, 5) = q(nnode, 5)

allocate(dq(nnode,5), stat = status)
if (status .ne. 0) then
   write(*,*)'Failed to allocate dq ',nnode, 5
   stop
end if
allocate(lams(nnode), stat = status)
if (status .ne. 0) then
   write(*,*)'Failed to allocate lams ',nnode
   stop
end if
allocate(D(nnode), stat = status)
if (status .ne. 0) then
   write(*,*)'Failed to allocate D ',nnode
   stop
end if
allocate(qp(nnode,5), stat = status)
if (status .ne. 0) then
   write(*,*)'Failed to allocate qp ',nnode, 5
   stop
end if
allocate(qmin(nnode,5), stat = status)
if (status .ne. 0) then
   write(*,*)'Failed to allocate qmin ',nnode, 5
   stop
end if
allocate(qmax(nnode,5), stat = status)
if (status .ne. 0) then
   write(*,*)'Failed to allocate qmax ',nnode, 5
   stop
end if

dq(:,:) = 0.0
qp(:,:) = 0.0
D(:) = 0.0
lams(:) = 0.0
qmin(:,:) = 0.0
qmax(:,:) = 0.0

call cpq2q0(5, nnode, q, q0)

! compute convective terms
! do m = 1, mstg
m = 1

   ! initialize residuals
   res = 0.0

   ! precompute the residuals in a rotational reference frame
   call rotat(nnode, cv, res, q, ynd, znd)

   ! calculate the state vector gradients
   call irecon(q, 6)

   ! calculate and store convective flux at every edge
   ! calculate and store residual for each node
   call iflux(nnode, xnd, ynd, znd, tx, ty, tz, nedge, ij_edge, sx, sy, &
        sz, ss, nbface, idbcs, np_bface, ip_bface, sbx, sby, sbz, sbs, &
        qb, cv, res)

   ! source terms of dual time-stepping scheme
   if (mtime .gt. 1) then
      call times(nnode, cv, res, q, q1, q2)
   end if

   ! compute viscous terms
   if (.not.invis) then
      call visrhs
      res = res + vres ! add viscous terms to RHS
   end if

   ! calculate the master / slave residual contribution
   call correct1(nnode, 5, res)
```

```
   ! local time step based on a constant CFL number
   !if (m .eq. 1) then
      call dtc(q, dtrd, cfl, emax, emay, emaz, ttime, lastiter)
   !end if

   ! multiply residuals by the local time step
   !call correct(nnode, 5, res, dtv, rk(m))

   ! performs implicit residual smoothing using Jacobi iteration
   if (irhsm .gt. 0) then
      call smooth(5, res, irhsm)
   end if

   ! reset the residuals at the wall, for viscous flows
   if (.not.invis) then
      call reset_wres(res, nnode, 5)
   end if

   ! update the state vector with m-stage Runge-Kutta step
   !call rkm(nnode, q0, res, q)

   ! convert variables into conservative before sweep calculations

   call switchpc(nnode, q, gamma, gm1, xgm1, debug, 1)

   ! Find initial dq' as performed in explicit method
   !********** Be aware of cfl numbers effect on dt and thus dq' !!!!!!!!!!!!!

   call expldq(nnode, res, dtv, cv, debug, dq, D)

   ! q' = q + dq' - Pseudo state variable n + 1/2

   qp(:,:) = q(:,1:5) + 0.5*dq(:,:)

   ! Find Implicit Operator D for pseudo time step n + 1/2

   call calcspec(qp,nnode,nedge,maxedge,xnd,ynd,znd,sxi,syi,szi,ij_edge, &
        edgnbr,cv,fmu,tur,gamma,gm1,invis,lamin,debug,lams)

   D(:) = D(:) + lams(:)

   if (impord .gt. 1) then
      call irecon(qp,5)

      if (lamin) then
         neq = 5
      else
         neq = 7
      end if

      call minlim(nnode,nedge,5,neq,nbface,igeom,qp,qb,qmin,qmax, &
           xnd,ynd,znd,tx,ty,tz,ux,uy,uz,vx,vy,vz,wx,wy,wz,px,py,pz, &
           phi_lim,e2,ij_edge,np_bface,ip_bface,idbcs,typlim,debug)
   end if

   ! Update state vector q using LU method

   call sweeps(nnode,nedge,maxedge,q,qp,dq,res,D,ij_edge,edgnbr, &
        lnbr,sx,sy,sz,ss,xnd,ynd,znd,sxi,syi,szi,omegax,rramp,gamma,zero, &
        debug,iter,nstep,istep1,fmu,tur,invis,lamin,qmin,qmax,tx,ty,tz, &
        ux,uy,uz,vx,vy,vz,wx,wy,wz,px,py,pz,phi_lim,neq,typlim,impord)

   ! update boundary conditions
   call bc(nnode, q, xnd, ynd, znd, nbface, np_bface, ip_bface, &
        idbcs, sbx, sby, sbz, sbs, qb, res)

   call update_q_mn2sn(q, nnode)

! end do            !End Runge-Kutta time marching


 ! compute the L2-norm of the residuals
 if (res_display) then
    call residl2(nnode, res, 5, rms, rmax, inode_rmax)
 end if

 if (monitormaxmach) then
    ! calculate max mach number value and cell location
    call max_mach_number(q, nnode, mach_max, max_mach_i)
    if (mach_max .gt. maxmachthreshold) then
       write (*,*) repeat("!+", 40)
       write (*,*) "Max Mach # ", mach_max, " exceeds threshold ", &
            maxmachthreshold
       write (*,*) repeat("!+", 40)
    end if
 end if
```

```
  deallocate(dq, stat = status)
  if (status .ne. 0) stop 'failure to deallocate memory in sweeps'
  deallocate(qp, stat = status)
  if (status .ne. 0) stop 'failure to deallocate memory in sweeps'
  deallocate(D, stat = status)
  if (status .ne. 0) stop 'failure to deallocate memory in sweeps'
  deallocate(lams, stat = status)
  if (status .ne. 0) stop 'failure to deallocate memory in sweeps'
  deallocate(qmin, stat = status)
  if (status .ne. 0) stop 'failure to deallocate memory in sweeps'
  deallocate(qmax, stat = status)
  if (status .ne. 0) stop 'failure to deallocate memory in sweeps'

  return
end subroutine lusgs
```

APPENDIX C

SOURCE CODE: SWITCHPC.F90

This is the source code which switches the state variables from conservative to primitive form and vice versa.

```
subroutine switchpc(nnode, q, gamma, gm1, xgm1, debug, flag)

  ! To switch q from conservative to primitive variables and vice versa

  ! flag = 1 : Primitive to Conservative
  ! flag = 2 : Conservative to Primitive

  ! jwc - 1.0 - 09/09/09 - To only have two transformations per iteration
  !                        which is close to rkm.f90 with one per iteration


  implicit none

  ! for precision

  integer,parameter :: dbl = kind(0.0d0)

  ! input

  integer,intent(in) :: nnode
  integer,intent(in) :: flag

  real(8),intent(in) :: gamma
  real(8),intent(in) :: gm1
  real(8),intent(in) :: xgm1

  logical,intent(in) :: debug

  ! output

  real(8),intent(inout) :: q(nnode,6)

  ! local

  integer :: i

  real(kind=dbl) :: r,u,v,w,p
  real(kind=dbl) :: rinv

  ! ----------------------------------------------------------------------
  if (debug) write(*,*) 'Calling switchpc'

  if (debug) write(*,*) gamma, gm1, xgm1


  ! switch from [r,u,v,w,p] to [r,ru,rv,rw,rE]
  if (flag .eq. 1) then
     do i = 1, nnode
        r = q(i,1) ! density
        u = q(i,2) ! u velocity
        v = q(i,3) ! v velocity
        w = q(i,4) ! w velocity
        p = q(i,5) ! pressure

        ! velocity conversion
        q(i,2) = r*u ! ru
        q(i,3) = r*v ! rv
        q(i,4) = r*w ! rw

        ! pressure conversion to energy
        q(i,5) = p*xgm1 + 0.5*(u*u + v*v + w*w)*r ! rE
     end do
  else ! flag .eq. 2: switch from [r,ru,rv,rw,rE] to [r,u,v,w,p]
     do i = 1, nnode
        r = q(i,1) ! density
        u = q(i,2) ! ru
```

```
        v = q(i,3) ! rv
        w = q(i,4) ! rw
        p = q(i,5) ! rE

        ! density conversion 1/r
        rinv = 1.0 / r

        ! velocity conversion rU/r
        q(i,2) = u*rinv ! u
        q(i,3) = v*rinv ! v
        q(i,4) = w*rinv ! w

        ! Energy conversion to pressure
        q(i,5) = gm1*(p - 0.5*rinv*(u*u + v*v + w*w)) ! pressure

        ! speed of sound squared ?????????
        q(i,6) = p * gamma * rinv
      end do
    end if


end subroutine switchpc
```

APPENDIX D

SOURCE CODE: EXPLDQ.F90

This is the source code which calculates an initial value for Δq_i at each iteration. This is where the cell volume part of the implicit operator is stored.

```
subroutine expldq(nnode, res, dtv, cv, debug, dq, D)

  implicit none

  ! for precision

  integer,parameter :: dbl = kind(0.0d0)

  ! input

  integer,intent(in) :: nnode

  real(8),intent(in) :: res(nnode,5)
  real(8),intent(in) :: dtv(nnode)
  real(8),intent(in) :: cv(nnode)

  logical,intent(in) :: debug

  !output

  real(kind=dbl),intent(inout) :: dq(nnode,5)
  real(kind=dbl),intent(inout) :: D(nnode)

  ! local

  integer :: i

  real(kind=dbl) :: alpha ! Coefficient to offset large cfl number?
  real(kind=dbl) :: beta
  real(kind=dbl) :: vol
  real(kind=dbl) :: dt

  ! ------------------------------------------------------------------

  if (debug) write(*,*) 'Calling expldq'

  alpha = 1.0
  beta = 1.0

  do i = 1, nnode
     dt = dtv(i)
     vol = cv(i)
     dq(i,:) = alpha*res(i,:)*dt/vol

     D(i) = beta*vol/dt
  end do


end subroutine expldq
```

APPENDIX E

SOURCE CODE: CALCSPEC.F90

This is the source code which calculates the spectral radius contribution to the implicit operator using the initial Δq_i computed in expldq.f90.

```
subroutine calcspec(q, nnode, nedge, maxedge, xnd, ynd, znd, sxi, syi, szi, &
     ij_edge, edgnbr, cv, fmu, tur, gamma, gm1, invis, lamin, debug, lams)

  ! Compute the spectral radius contribution to the implicit operator for each
  ! node

  implicit none

  ! for precision

  integer,parameter :: dbl = kind(0.0d0)

  ! input

  integer,intent(in) :: nnode
  integer,intent(in) :: nedge
  integer,intent(in) :: maxedge
  integer,intent(in) :: ij_edge(nedge,2)
  integer,intent(in) :: edgnbr(nnode,maxedge+1)

  real(8),intent(in) :: gamma
  real(8),intent(in) :: gm1
  real(8),intent(in) :: q(nnode,6)
  real(8),intent(in) :: xnd(nnode)
  real(8),intent(in) :: ynd(nnode)
  real(8),intent(in) :: znd(nnode)
  real(8),intent(in) :: sxi(nnode)
  real(8),intent(in) :: syi(nnode)
  real(8),intent(in) :: szi(nnode)
  real(8),intent(in) :: cv(nnode)
  real(8),intent(in) :: fmu(nnode)
  real(8),intent(in) :: tur(nnode)

  logical,intent(in) :: debug
  logical,intent(in) :: invis
  logical,intent(in) :: lamin

  ! output

  real(kind=dbl),intent(inout) :: lams(nnode)

  ! local

  integer :: i, inode, jnode, iedge, jedge

  real(kind=dbl) :: ri, ui, vi, wi, pi
  real(kind=dbl) :: rj, uj, vj, wj, pj
  real(kind=dbl) :: xi, yi, zi
  real(kind=dbl) :: xj, yj, zj
  real(kind=dbl) :: rinv
  real(kind=dbl) :: lam, lami, lamj
  real(kind=dbl) :: axi, ayi, azi, axj, ayj, azj, voli, volj

  ! ---------------------------------------------------------------------

  if (debug) write(*,*) 'Calling calcspec'

  lam = 0.0

  ! Loop through all nodes to compute sum of spectral radii
  do i = 1,nnode

     ! Go in order of mesh
     inode = edgnbr(i,1)

     if (inode .eq. 11034 .AND. debug) write(*,*)'calcspec'
```

```
   ri = q(inode,1)
   ui = q(inode,2)
   vi = q(inode,3)
   wi = q(inode,4)
   pi = q(inode,5)

   xi = xnd(inode)
   yi = ynd(inode)
   zi = znd(inode)

   axi = sxi(inode)
   ayi = syi(inode)
   azi = szi(inode)

   voli = cv(inode)

   rinv = 1.0/ri
   ui = ui*rinv
   vi = vi*rinv
   wi = wi*rinv
   pi = gm1*(pi - 0.5*ri*(ui*ui + vi*vi + wi*wi))

   if (ri .lt. 0.0 .OR. pi .lt. 0.0) then
      write(*,*) 'ri, pi lt zero: ', inode
      stop
   end if

   ! Obtain edges and nodes connected to inode
   do iedge = 2, maxedge+1

      jedge = edgnbr(i,iedge)

      if (jedge .ne. 0) then

         jnode = ij_edge(jedge,2)

         if (jnode .eq. inode) then
            jnode = ij_edge(jedge,1)
         end if

         rj = q(jnode,1)
         uj = q(jnode,2)
         vj = q(jnode,3)
         wj = q(jnode,4)
         pj = q(jnode,5)

         xj = xnd(jnode)
         yj = ynd(jnode)
         zj = znd(jnode)

         axj = sxi(jnode)
         ayj = syi(jnode)
         azj = szi(jnode)

         volj = cv(jnode)

         rinv = 1.0/rj
         uj = uj*rinv
         vj = vj*rinv
         wj = wj*rinv
         pj = gm1*(pj - 0.5*rj*(uj*uj + vj*vj + wj*wj))

         if (rj .lt. 0.0 .OR. pj .lt. 0.0) then
            write(*,*) 'rj, pj lt zero: ', jnode
            stop
         end if

         call spectrad(nnode,inode,jnode,ri,ui,vi,wi,pi,xi,yi,zi, &
              xj,yj,zj,axi,ayi,azi,voli,fmu(inode),tur(inode),gamma, &
              invis,lamin,debug,lami)
         call spectrad(nnode,jnode,inode,rj,uj,vj,wj,pj,xj,yj,zj, &
              xi,yi,zi,axj,ayj,azj,volj,fmu(jnode),tur(jnode),gamma, &
              invis,lamin,debug,lamj)

         lam = 0.5*(lami + lamj)

         lams(inode) = lams(inode) + 0.5*lam
      else

         lams(inode) = lams(inode) + 0.0

      end if
   end do
end do
```

```
end subroutine calcspec
```

APPENDIX F

SOURCE CODE: SPECTRAD.F90

This is the source code which computes the spectral radius using the updates state variables during the forward and backward sweeps.

```
subroutine spectrad(nnode, inode, jnode, ri, ui, vi, wi, pi, xi, yi, zi, &
     xj, yj, zj, ax, ay, az, cv, fmu, tur, gamma, invis, lamin, debug, lam)

  ! calculate the spectral radius for each interface between cells

  !use mesh_vars,      only: sxi, syi, szi !, ss
  use gasprop,        only: prl, prt

  implicit none

  ! variable type

  integer,parameter::dbl=kind(0.0d0)

  ! input

  integer,intent(in) :: nnode
  integer,intent(in) :: inode
  integer,intent(in) :: jnode

  real(8),intent(in) :: gamma
  real(8),intent(in) :: ri
  real(8),intent(in) :: ui
  real(8),intent(in) :: vi
  real(8),intent(in) :: wi
  real(8),intent(in) :: pi
  real(8),intent(in) :: fmu
  real(8),intent(in) :: tur
  real(8),intent(in) :: xi
  real(8),intent(in) :: yi
  real(8),intent(in) :: zi
  real(8),intent(in) :: xj
  real(8),intent(in) :: yj
  real(8),intent(in) :: zj
  real(8),intent(in) :: ax
  real(8),intent(in) :: ay
  real(8),intent(in) :: az
  real(8),intent(in) :: cv

  logical,intent(in) :: debug
  logical,intent(in) :: invis
  logical,intent(in) :: lamin

  ! output

  real(kind=dbl),intent(out) :: lam

  ! local

  !integer :: iedge, i, j

  real(kind=dbl) :: u,v,w
  real(kind=dbl) :: dx,dy,dz
  real(kind=dbl) :: nx,ny,nz,R2
  real(kind=dbl) :: area !, ax, ay, az
  real(kind=dbl) :: c,vn,den
  real(kind=dbl) :: mu,mut
  real(kind=dbl) :: visc

  ! ---------------------------------------------------------------------

  if (debug) write(*,*) 'Calling spectrad'

  dx = xj - xi
  dy = yj - yi
  dz = zj - zi
```

```
  R2 = sqrt(dx*dx + dy*dy + dz*dz)

  R2 = 1.0/R2

  nx = dx*R2
  ny = dy*R2
  nz = dz*R2

  !ax = sxi(inode)
  !ay = syi(inode)
  !az = szi(inode)

  if (invis) then
     mu = 0.0
     mut = 0.0
  else
     mu = fmu
     !if (inode .eq. 1) write(*,*) mu

     if (.not.lamin) then
        mut = tur
     else
        mut = 0.0
     end if
  end if

  !mu = 2.0 * (mu + mut)    ! chen and wang

  !vn = abs(ui*nx + vi*ny + wi*nz)
  u = abs(ui)   !ui*nx
  v = abs(vi)   !vi*ny
  w = abs(wi)   !wi*nz
  !vn = u*u + v*v + w*w
  !vn = sqrt(vn)

  c = gamma*pi/ri
  if (c < 0.0) then
     write(*,*)'sqrt negative number, spect rad'
     write(*,*)'inode, jnode: ',inode, jnode
     write(*,10)ri,ui,vi,wi,pi
     10 format(1x,5(ES12.5,2x))
     stop
  end if
  c = sqrt(c)

  u = (u + c)*ax
  v = (v + c)*ay
  w = (w + c)*az

  vn = u + v + w

  !den = ri*abs(dx*nx + dy*ny + dz*nz)  ! chen and wang
  den = ri*cv

  den = 1.0/den
  visc = (mu/prl) + (mut/prt)
  area = ax*ax + ay*ay + az*az

  !lam = vn + (den*mu)  ! chen and wang
  lam = vn + den*gamma*visc*area


end subroutine spectrad
```

APPENDIX G

SOURCE CODE: SWEEPS.F90

This is the source code that employs the forward and backward sweep techniques used in the LUSGS integration scheme.

```
subroutine sweeps(nnode,nedge,maxedge,q,qp,dq,res,D,ij_edge,edgnbr, &
     lnbr,sx,sy,sz,ss,xnd,ynd,znd,sxi,syi,szi,omegax,rramp,gamma,zero, &
     debug,iter,nstep,istep1,fmu,tur,invis,lamin,qmin,qmax,rx,ry,rz, &
     ux,uy,uz,vx,vy,vz,wx,wy,wz,px,py,pz,phi_lim,neq,typlim,impord)

  ! Uses switchpc to convert variables from primitive to conservative before
  ! calculations are performed.  Variables are converted into primitive using
  ! local parameters for inviscid flux calculations

  ! Call spectradV4 and DcalcV3 when needed in code to distinguish between
  ! inode and jnode usage.

  use mesh_vars,     only: cv, dtv     ! for pure debugging: rid of when done!
  use gasprop,       only: gm1, xgm1   ! for conversion between primitive
                                       ! and conservative variables
  !use ibase,         only: iorder      ! for flux calculations

  implicit none

  ! for precision

  integer,parameter :: dbl = kind(0.0d0)

  !input

  integer,intent(in) :: nnode
  integer,intent(in) :: nedge
  integer,intent(in) :: maxedge
  integer,intent(in) :: ij_edge(nedge,2)
  integer,intent(in) :: edgnbr(nnode,maxedge+1)
  integer,intent(in) :: lnbr(nnode)
  integer,intent(in) :: iter
  integer,intent(in) :: nstep
  integer,intent(in) :: istep1
  integer,intent(in) :: neq
  integer,intent(in) :: typlim
  integer,intent(in) :: impord

  real(8),intent(in) :: omegax
  real(8),intent(in) :: rramp
  real(8),intent(in) :: zero
  real(8),intent(in) :: gamma
!  real(8),intent(in) :: res(nnode,5)
  real(kind=dbl),intent(in) :: D(nnode)
  real(8),intent(in) :: sx(nedge)
  real(8),intent(in) :: sy(nedge)
  real(8),intent(in) :: sz(nedge)
  real(8),intent(in) :: ss(nedge)
  real(8),intent(in) :: xnd(nnode)
  real(8),intent(in) :: ynd(nnode)
  real(8),intent(in) :: znd(nnode)
  real(8),intent(in) :: sxi(nnode)
  real(8),intent(in) :: syi(nnode)
  real(8),intent(in) :: szi(nnode)
  real(8),intent(in) :: fmu(nnode)
  real(8),intent(in) :: tur(nnode)
  real(kind=dbl),intent(in) :: qmin(nnode,5)
  real(kind=dbl),intent(in) :: qmax(nnode,5)
  real(8),intent(in) :: rx(nnode)
  real(8),intent(in) :: ry(nnode)
  real(8),intent(in) :: rz(nnode)
  real(8),intent(in) :: ux(nnode)
  real(8),intent(in) :: uy(nnode)
  real(8),intent(in) :: uz(nnode)
  real(8),intent(in) :: vx(nnode)
  real(8),intent(in) :: vy(nnode)
```

```
  real(8),intent(in) :: vz(nnode)
  real(8),intent(in) :: wx(nnode)
  real(8),intent(in) :: wy(nnode)
  real(8),intent(in) :: wz(nnode)
  real(8),intent(in) :: px(nnode)
  real(8),intent(in) :: py(nnode)
  real(8),intent(in) :: pz(nnode)
  real(8),intent(in) :: phi_lim(nnode,neq)

  logical,intent(in) :: debug
  logical,intent(in) :: invis
  logical,intent(in) :: lamin

  !output

  real(8),intent(inout) :: q(nnode,6)
  real(kind=dbl),intent(inout) :: qp(nnode,5)
  real(kind=dbl),intent(inout) :: dq(nnode,5)
  real(8),intent(inout) :: res(nnode,5)

  !local

  integer :: inode
  integer :: iedge
  integer :: jedge
  integer :: i,j,m
  integer :: jnode
  integer :: nbrs
  integer :: status
  integer :: k, kmax
  integer :: blid
  integer :: cnode
  integer :: dir

  real(kind=dbl) :: area,ax,ay,az
  real(kind=dbl) :: axi,ayi,azi,axj,ayj,azj
  real(kind=dbl) :: lmax
  real(kind=dbl) :: omegax1,vn,vgy,vgz,yc,zc
  real(kind=dbl) :: ri,ui,vi,wi,pi
!  real(kind=dbl) :: dr,du,dv,dw,dE
  real(kind=dbl) :: rinv
  real(kind=dbl) :: dri,dui,dvi,dwi,dpi
  real(kind=dbl) :: rj,uj,vj,wj,pj
  real(kind=dbl) :: drj,duj,dvj,dwj,dpj
  real(kind=dbl) :: xi,yi,zi
  real(kind=dbl) :: xj,yj,zj
  real(kind=dbl) :: dx,dy,dz
  real(kind=dbl) :: den
  real(kind=dbl) :: lam,lami,lamj
  real(kind=dbl) :: deltri,deltui,deltvi,deltwi,deltpi
  real(kind=dbl) :: deltrj,deltuj,deltvj,deltwj,deltpj
  real(kind=dbl) :: phi_i1,phi_i2,phi_i3,phi_i4,phi_i5
  real(kind=dbl) :: phi_j1,phi_j2,phi_j3,phi_j4,phi_j5
  real(kind=dbl) :: qmin_i1,qmin_i2,qmin_i3,qmin_i4,qmin_i5
  real(kind=dbl) :: qmin_j1,qmin_j2,qmin_j3,qmin_j4,qmin_j5
  real(kind=dbl) :: qmax_i1,qmax_i2,qmax_i3,qmax_i4,qmax_i5
  real(kind=dbl) :: qmax_j1,qmax_j2,qmax_j3,qmax_j4,qmax_j5
  real(kind=dbl),allocatable :: fl(:)
  real(kind=dbl),allocatable :: dfl(:)
  real(kind=dbl),allocatable :: summ(:)
  !real(kind=dbl),allocatable :: dq(:,:)
  !real(kind=dbl),allocatable :: lams(:)
  !real(kind=dbl),allocatable :: D(:)
  !real(kind=dbl),allocatable :: qp(:,:)

  real(kind=dbl) :: Bl(5,5)
  real(kind=dbl) :: dq0(2,5)
  ! --------------------------------------------------------------------

  if (debug) write(*,*) 'Calling sweeps'

  allocate(fl(5), stat = status)
  if (status .ne. 0) then
     write(*,*) 'Failed to allocate fl ',5
     stop
  end if
  allocate(dfl(5), stat = status)
  if (status .ne. 0) then
     write(*,*) 'Failed to allocate dfl ',5
     stop
  end if
  allocate(summ(5), stat = status)
  if (status .ne. 0) then
     write(*,*) 'Failed to allocate summ ',5
     stop
  end if
```

```
omegax1 = omegax * rramp
if (omegax1 .eq. zero) then
   vn = zero
end if

summ(:) = 0.0
fl(:) = 0.0
dfl(:) = 0.0

dir = 1
lam = 0.0

do j=1,5
   do k=1,5
      Bl(j,k) = 0.0
   end do
end do

cnode = 1
blid = 234 + nstep
kmax = 1

if (iter .eq. istep1 .AND. debug) then
   open(unit=blid,file='bl.dat',action='write',position='append')
end if

write(blid,*) 'Iteration: ',iter
write(blid,*)

! Inner Sweeping

do k=1,kmax

   ! Forward Sweeping
   if (debug) write(*,*) 'forward sweeping'
   do i = 1, nnode
      inode = edgnbr(i,1)

      if (inode .eq. cnode .AND. debug) write(*,*) 'forward sweep, lam'

      ! Using state variable as static per iteration
      ri = qp(inode,1)   ! r
      ui = qp(inode,2)   ! ru
      vi = qp(inode,3)   ! rv
      wi = qp(inode,4)   ! rw
      pi = qp(inode,5)   ! rE

      ! Prepare variables for flux calculations: primitive form

      rinv = 1.0/ri
      ui = rinv * ui
      vi = rinv * vi
      wi = rinv * wi
      pi = gm1*(pi - 0.5 * ri * (ui*ui + vi*vi + wi*wi))

      xi = xnd(inode)
      yi = ynd(inode)
      zi = znd(inode)

      axi = sxi(inode)
      ayi = syi(inode)
      azi = szi(inode)

      dq0(1,1) = ri
      dq0(1,2) = ui
      dq0(1,3) = vi
      dq0(1,4) = wi
      dq0(1,5) = pi

      dri = ri
      dui = ui
      dvi = vi
      dwi = wi
      dpi = pi

      nbrs = lnbr(i)

      if (debug) write(*,*) 'inode, nbrs: ',inode, nbrs, maxedge

      do iedge = 2, nbrs+1, 1 ! Lower Summation

         if (debug) write(*,*) 'Lower Summation'

         jedge = edgnbr(i,iedge)
         jnode = ij_edge(jedge,2)
         dir = 1
```

```
if (jnode .eq. inode) then
   jnode = ij_edge(jedge,1)
   dir = 2
end if

! Using state variable as static per iteration
rj = qp(jnode,1)   ! r
uj = qp(jnode,2)   ! ru
vj = qp(jnode,3)   ! rv
wj = qp(jnode,4)   ! rw
pj = qp(jnode,5)   ! rE

xj = xnd(jnode)
yj = ynd(jnode)
zj = znd(jnode)

axj = sxi(jnode)
ayj = syi(jnode)
azj = szi(jnode)

if (rj .lt. 0.0) write(*,*)' rj = ',rj
if (pj .lt. 0.0) write(*,*)' pj = ',pj

! Q _n+1 = Q + dq'

drj = rj + 0.5*dq(jnode,1)   ! r
duj = uj + 0.5*dq(jnode,2)   ! ru
dvj = vj + 0.5*dq(jnode,3)   ! rv
dwj = wj + 0.5*dq(jnode,4)   ! rw
dpj = pj + 0.5*dq(jnode,5)   ! rE

if (drj .lt. 0.0) write(*,*)' rj, dq = ', rj,' ',drj-rj, jnode
if (dpj .lt. 0.0) write(*,*)' pj, dq = ', pj,' ',dpj-pj, jnode

! Prepare variables for flux calculations: primitive form

rinv = 1.0/rj
uj = rinv * uj
vj = rinv * vj
wj = rinv * wj
pj = gm1*(pj - 0.5 * rj * (uj*uj + vj*vj + wj*wj))

dq0(2,1) = rj
dq0(2,2) = uj
dq0(2,3) = vj
dq0(2,4) = wj
dq0(2,5) = pj

rinv = 1.0/drj
duj = rinv * duj
dvj = rinv * dvj
dwj = rinv * dwj
dpj = gm1*(dpj - 0.5 * drj * (duj*duj + dvj*dvj + dwj*dwj))

! Higher Spatial Order Q Preparation

if (impord .gt. 1) then
   dx = xj - xi
   dy = yj - yi
   dz = zj - zi

   phi_i1 = phi_lim(inode,1)
   phi_i2 = phi_lim(inode,2)
   phi_i3 = phi_lim(inode,3)
   phi_i4 = phi_lim(inode,4)
   phi_i5 = phi_lim(inode,5)

   qmin_i1 = qmin(inode,1)
   qmin_i2 = qmin(inode,2)
   qmin_i3 = qmin(inode,3)
   qmin_i4 = qmin(inode,4)
   qmin_i5 = qmin(inode,5)

   qmax_i1 = qmax(inode,1)
   qmax_i2 = qmax(inode,2)
   qmax_i3 = qmax(inode,3)
   qmax_i4 = qmax(inode,4)
   qmax_i5 = qmax(inode,5)

   deltri = rx(inode)*dx + ry(inode)*dy + rz(inode)*dz
   deltui = ux(inode)*dx + uy(inode)*dy + uz(inode)*dz
   deltvi = vx(inode)*dx + vy(inode)*dy + vz(inode)*dz
   deltwi = wx(inode)*dx + wy(inode)*dy + wz(inode)*dz
   deltpi = px(inode)*dx + py(inode)*dy + pz(inode)*dz

   if (typlim .ne. 3) then
      dx = -1.0*dx
```

```
      dy = -1.0*dy
      dz = -1.0*dz
   end if

   phi_j1 = phi_lim(jnode,1)
   phi_j2 = phi_lim(jnode,2)
   phi_j3 = phi_lim(jnode,3)
   phi_j4 = phi_lim(jnode,4)
   phi_j5 = phi_lim(jnode,5)

   qmin_j1 = qmin(jnode,1)
   qmin_j2 = qmin(jnode,2)
   qmin_j3 = qmin(jnode,3)
   qmin_j4 = qmin(jnode,4)
   qmin_j5 = qmin(jnode,5)

   qmax_j1 = qmax(jnode,1)
   qmax_j2 = qmax(jnode,2)
   qmax_j3 = qmax(jnode,3)
   qmax_j4 = qmax(jnode,4)
   qmax_j5 = qmax(jnode,5)

   deltrj = rx(jnode)*dx + ry(jnode)*dy + rz(jnode)*dz
   deltuj = ux(jnode)*dx + uy(jnode)*dy + uz(jnode)*dz
   deltvj = vx(jnode)*dx + vy(jnode)*dy + vz(jnode)*dz
   deltwj = wx(jnode)*dx + wy(jnode)*dy + wz(jnode)*dz
   deltpj = px(jnode)*dx + py(jnode)*dy + pz(jnode)*dz

   call flxio2(ri,ui,vi,wi,pi,rj,uj,vj,wj,pj,dx,dy,dz, &
        deltri,deltui,deltvi,deltwi,deltpi,deltrj,deltuj, &
        deltvj,deltwj,deltpj,phi_i1,phi_i2,phi_i3,phi_i4, &
        phi_i5,phi_j1,phi_j2,phi_j3,phi_j4,phi_j5,qmin_i1, &
        qmin_i2,qmin_i3,qmin_i4,qmin_i5,qmin_j1,qmin_j2, &
        qmin_j3,qmin_j4,qmin_j5,qmax_i1,qmax_i2,qmax_i3, &
        qmax_i4,qmax_i5,qmax_j1,qmax_j2,qmax_j3,qmax_j4, &
        qmax_j5,typlim,debug)

   call flxio2(dri,dui,dvi,dwi,dpi,drj,duj,dvj,dwj,dpj,dx,dy,dz, &
        deltri,deltui,deltvi,deltwi,deltpi,deltrj,deltuj, &
        deltvj,deltwj,deltpj,phi_i1,phi_i2,phi_i3,phi_i4, &
        phi_i5,phi_j1,phi_j2,phi_j3,phi_j4,phi_j5,qmin_i1, &
        qmin_i2,qmin_i3,qmin_i4,qmin_i5,qmin_j1,qmin_j2, &
        qmin_j3,qmin_j4,qmin_j5,qmax_i1,qmax_i2,qmax_i3, &
        qmax_i4,qmax_i5,qmax_j1,qmax_j2,qmax_j3,qmax_j4, &
        qmax_j5,typlim,debug)

end if

! Calculate Rotating Velocity

if (omegax1 .ne. zero) then
   yc = 0.5 * (yi + yj)
   zc = 0.5 * (zi + zj)
   vgy = -omegax1 * zc
   vgz = omegax1 * yc
   vn = -sy(jedge) * vgy - sz(jedge) * vgz
end if

area = abs(ss(jedge))
ax = sx(jedge)
ay = sy(jedge)
az = sz(jedge)

if (dir .eq. 1) then
   ! for left state being inode (i), right state being jnode (j)
   call fhat6(fl,ax,ay,az,vn,ri,ui,vi,wi,pi,rj,uj,vj,wj,pj,lmax,i)

   call fhat6(dfl,ax,ay,az,vn,dri,dui,dvi,dwi,dpi, &
        drj,duj,dvj,dwj,dpj,lmax,i)

else

   ! for left state being jnode (j), right state being inode (i)
   call fhat6(fl,-ax,-ay,-az,vn,ri,ui,vi,wi,pi, &
        rj,uj,vj,wj,pj,lmax,i)

   call fhat6(dfl,-ax,-ay,-az,vn,dri,dui,dvi,dwi,dpi, &
        drj,duj,dvj,dwj,dpj,lmax,i)

   fl(:) = -1.0*fl(:)
   dfl(:) = -1.0*dfl(:)
   area = -1.0*area
end if

call spectrad(nnode,inode,jnode,ri,ui,vi,wi,pi,xi,yi,zi, &
     xj,yj,zj,axi,ayi,azi,cv(inode),fmu(inode),tur(inode),gamma, &
     invis,lamin,debug,lami)
```

```
      call spectrad(nnode,inode,jnode,rj,uj,vj,wj,pj,xj,yj,zj, &
           xi,yi,zi,axj,ayj,azj,cv(inode),fmu(jnode),tur(jnode),gamma, &
           invis,lamin,debug,lamj)

      lam = (lami + lamj)*0.5

      summ(:) = summ(:) + (dfl(:) - fl(:))*area - lam*dq(jnode,:)*area

      fl(:) = 0.0
      dfl(:) = 0.0

   end do

   den = D(inode)

   do j=1,5
      Bl(j,j) = den
   end do

   den = 1.0/den

   ! solution of dqs in conservative form

   dq(inode,:) = den * (res(inode,:) - 0.5 * summ(:))

   if (inode .eq. cnode) then
      write(blid,*) 'Forward Sweep, Lower Summation'
      write(blid,*) 'iter: ',iter
      write(blid,*) 'k= ',k
      write(blid,*) 'inode: ',inode
      write(blid,*) 'dt: ',dtv(inode)
      write(blid,*) 'vol: ',cv(inode)
      write(blid,11) (q(inode,j),j=1,5)
      write(blid,11) (dq(inode,j),j=1,5)
      11 format (2x,5(ES14.7,2x))
      write(blid,11) (res(inode,j),j=1,5)
      write(blid,11) (summ(j),j=1,5)
      write(blid,*)
      do m=1,5
         write(blid,11) (Bl(m,j),j=1,5)
      end do
      write(blid,*)
   end if

   qp(inode,:) = q(inode,1:5) + 0.5*dq(inode,:)

   summ(:) = 0.0

end do

! Backward Sweep

summ(:) = 0.0

do i = nnode, 1, -1

   if (debug) write(*,*) 'Backward Sweep'

   inode = edgnbr(i,1)

   if (inode .eq. cnode .AND. debug) write(*,*) 'backward sweep, lam'

   ! Using state variable as static per iteration
   ri = qp(inode,1)   ! r
   ui = qp(inode,2)   ! ru
   vi = qp(inode,3)   ! rv
   wi = qp(inode,4)   ! rw
   pi = qp(inode,5)   ! rE

   nbrs = lnbr(i)

   ! Prepare variables for flux calculations: primitive form

   rinv = 1.0/ri
   ui = rinv * ui
   vi = rinv * vi
   wi = rinv * wi
   pi = gm1*(pi - 0.5 * ri * (ui*ui + vi*vi + wi*wi))

   xi = xnd(inode)
   yi = ynd(inode)
   zi = znd(inode)

   axi = sxi(inode)
   ayi = syi(inode)
   azi = szi(inode)
```

```
dq0(1,1) = ri
dq0(1,2) = ui
dq0(1,3) = vi
dq0(1,4) = wi
dq0(1,5) = pi

dri = ri
dui = ui
dvi = vi
dwi = wi
dpi = pi

! Upper Summation
do iedge = nbrs+2, maxedge+1,1

   if (debug) write(*,*) 'Upper Summation'

   jedge = edgnbr(i,iedge)

   if (jedge .ne. 0) then

      jnode = ij_edge(jedge,2)
      dir = 1

      if (jnode .eq. inode) then
         jnode = ij_edge(jedge,1)
         dir = 2
      end if

      ! Using state variavble as static per iteration
      rj = qp(jnode,1)   ! r
      uj = qp(jnode,2)   ! ru
      vj = qp(jnode,3)   ! rv
      wj = qp(jnode,4)   ! rw
      pj = qp(jnode,5)   ! rE

      xj = xnd(jnode)
      yj = ynd(jnode)
      zj = znd(jnode)

      axj = sxi(jnode)
      ayj = syi(jnode)
      azj = szi(jnode)

      ! Q _n+1 = Q + dq'

      drj = rj + 0.5*dq(jnode,1)   ! r

      if (drj .lt. zero) then
         write(*,*)'drho is negative'
         write(*,*)'rj ',rj,' drho = ',dq(jnode,1)
         write(*,*)uj, vj,wj, pj
         write(*,*)dq(jnode,:)
         stop
      end if

      duj = uj + 0.5*dq(jnode,2)   ! ru
      dvj = vj + 0.5*dq(jnode,3)   ! rv
      dwj = wj + 0.5*dq(jnode,4)   ! rw
      dpj = pj + 0.5*dq(jnode,5)   ! rE

      ! Prepare variables for flux calculations: primitive form

      rinv = 1.0/rj
      uj = rinv * uj
      vj = rinv * vj
      wj = rinv * wj
      pj = gm1*(pj - 0.5 * rj * (uj*uj + vj*vj + wj*wj))

      dq0(2,1) = rj
      dq0(2,2) = uj
      dq0(2,3) = vj
      dq0(2,4) = wj
      dq0(2,5) = pj

      rinv = 1.0/drj
      duj = rinv * duj
      dvj = rinv * dvj
      dwj = rinv * dwj
      dpj = gm1*(dpj - 0.5 * drj * (duj*duj + dvj*dvj + dwj*dwj))

      ! Higher Spatial Order Q Preparation

      if (impord .gt. 1) then
         dx = xj - xi
         dy = yj - yi
         dz = zj - zi
```

```
   phi_i1 = phi_lim(inode,1)
   phi_i2 = phi_lim(inode,2)
   phi_i3 = phi_lim(inode,3)
   phi_i4 = phi_lim(inode,4)
   phi_i5 = phi_lim(inode,5)

   qmin_i1 = qmin(inode,1)
   qmin_i2 = qmin(inode,2)
   qmin_i3 = qmin(inode,3)
   qmin_i4 = qmin(inode,4)
   qmin_i5 = qmin(inode,5)

   qmax_i1 = qmax(inode,1)
   qmax_i2 = qmax(inode,2)
   qmax_i3 = qmax(inode,3)
   qmax_i4 = qmax(inode,4)
   qmax_i5 = qmax(inode,5)

   deltri = rx(inode)*dx + ry(inode)*dy + rz(inode)*dz
   deltui = ux(inode)*dx + uy(inode)*dy + uz(inode)*dz
   deltvi = vx(inode)*dx + vy(inode)*dy + vz(inode)*dz
   deltwi = wx(inode)*dx + wy(inode)*dy + wz(inode)*dz
   deltpi = px(inode)*dx + py(inode)*dy + pz(inode)*dz

   if (typlim .ne. 3) then
      dx = -1.0*dx
      dy = -1.0*dy
      dz = -1.0*dz
   end if

   phi_j1 = phi_lim(jnode,1)
   phi_j2 = phi_lim(jnode,2)
   phi_j3 = phi_lim(jnode,3)
   phi_j4 = phi_lim(jnode,4)
   phi_j5 = phi_lim(jnode,5)

   qmin_j1 = qmin(jnode,1)
   qmin_j2 = qmin(jnode,2)
   qmin_j3 = qmin(jnode,3)
   qmin_j4 = qmin(jnode,4)
   qmin_j5 = qmin(jnode,5)

   qmax_j1 = qmax(jnode,1)
   qmax_j2 = qmax(jnode,2)
   qmax_j3 = qmax(jnode,3)
   qmax_j4 = qmax(jnode,4)
   qmax_j5 = qmax(jnode,5)

   deltrj = rx(jnode)*dx + ry(jnode)*dy + rz(jnode)*dz
   deltuj = ux(jnode)*dx + uy(jnode)*dy + uz(jnode)*dz
   deltvj = vx(jnode)*dx + vy(jnode)*dy + vz(jnode)*dz
   deltwj = wx(jnode)*dx + wy(jnode)*dy + wz(jnode)*dz
   deltpj = px(jnode)*dx + py(jnode)*dy + pz(jnode)*dz

   call flxio2(ri,ui,vi,wi,pi,rj,uj,vj,wj,pj,dx,dy,dz, &
        deltri,deltui,deltvi,deltwi,deltpi,deltrj,deltuj, &
        deltvj,deltwj,deltpj,phi_i1,phi_i2,phi_i3,phi_i4, &
        phi_i5,phi_j1,phi_j2,phi_j3,phi_j4,phi_j5,qmin_i1, &
        qmin_i2,qmin_i3,qmin_i4,qmin_i5,qmin_j1,qmin_j2, &
        qmin_j3,qmin_j4,qmin_j5,qmax_i1,qmax_i2,qmax_i3, &
        qmax_i4,qmax_i5,qmax_j1,qmax_j2,qmax_j3,qmax_j4, &
        qmax_j5,typlim,debug)

   call flxio2(dri,dui,dvi,dwi,dpi,drj,duj,dvj,dwj,dpj,dx,dy,dz, &
        deltri,deltui,deltvi,deltwi,deltpi,deltrj,deltuj, &
        deltvj,deltwj,deltpj,phi_i1,phi_i2,phi_i3,phi_i4, &
        phi_i5,phi_j1,phi_j2,phi_j3,phi_j4,phi_j5,qmin_i1, &
        qmin_i2,qmin_i3,qmin_i4,qmin_i5,qmin_j1,qmin_j2, &
        qmin_j3,qmin_j4,qmin_j5,qmax_i1,qmax_i2,qmax_i3, &
        qmax_i4,qmax_i5,qmax_j1,qmax_j2,qmax_j3,qmax_j4, &
        qmax_j5,typlim,debug)

end if

! Calculate Rotating Velocity

if (omegax1 .ne. zero) then
   yc = 0.5 * (yi + yj)
   zc = 0.5 * (zi + zj)
   vgy = -omegax1 * zc
   vgz = omegax1 * yc
   vn = -sy(jedge) * vgy - sz(jedge) * vgz
end if

area = abs(ss(jedge))
ax = sx(jedge)
```

```
            ay = sy(jedge)
            az = sz(jedge)

            if (dir .eq. 1) then
               ! for left state being inode (i), right state being jnode (j)
               call fhat6(fl,ax,ay,az,vn,ri,ui,vi,wi,pi,rj,uj,vj,wj,pj,lmax,i)

               call fhat6(dfl,ax,ay,az,vn,dri,dui,dvi,dwi,dpi, &
                    drj,duj,dvj,dwj,dpj,lmax,i)
            else
               ! for left state being jnode (j), right state being inode (i)
               call fhat6(fl,-ax,-ay,-az,vn,ri,ui,vi,wi,pi, &
                    rj,uj,vj,wj,pj,lmax,i)

               call fhat6(dfl,-ax,-ay,-az,vn,dri,dui,dvi,dwi,dpi, &
                    drj,duj,dvj,dwj,dpj,lmax,i)

               fl(:) = -1.0*fl(:)
               dfl(:) = -1.0*dfl(:)
               area = -1.0*area
            end if

            call spectrad(nnode,inode,jnode,ri,ui,vi,wi,pi,xi,yi,zi, &
                 xj,yj,zj,axi,ayi,azi,cv(inode),fmu(inode),tur(inode),gamma, &
                 invis,lamin,debug,lami)
            call spectrad(nnode,inode,jnode,rj,uj,vj,wj,pj,xj,yj,zj, &
                 xi,yi,zi,axj,ayj,azj,cv(inode),fmu(jnode),tur(jnode),gamma, &
                 invis,lamin,debug,lamj)

            lam = (lami + lamj)*0.5

            summ(:) = summ(:) + (dfl(:) - fl(:))*area - lam*dq(jnode,:)*area

            dfl(:) = 0.0
            fl(:) = 0.0

         else
            summ(:) = summ(:) + 0.0
         end if

      end do

      den = D(inode)

      do j=1,5
         Bl(j,j) = den
      end do

      den = 1.0/den

      ! solve for dq in conservative form

      dq(inode,:) = dq(inode,:) - 0.5 * den * (summ(:))

      if (inode .eq. cnode) then
         write(blid,*) 'Backward Sweep, Upper Summation'
         write(blid,*) 'iter: ',iter
         write(blid,*) 'k= ',k
         write(blid,*) 'inode: ',inode
         write(blid,*) 'dt: ', dtv(inode)
         write(blid,*) 'vol: ', cv(inode) !*6.0
         write(blid,11) (q(inode,j),j=1,5)
         write(blid,11) (dq(inode,j),j=1,5)
         !11 format (2x,5(ES14.7,2x))
         write(blid,11) (res(inode,j),j=1,5)
         write(blid,11) (summ(j),j=1,5)
         write(blid,*)
         do m=1,5
            write(blid,11) (Bl(m,j),j=1,5)
         end do
         write(blid,*)
      end if

      qp(inode,:) = q(inode,1:5) + 0.5*dq(inode,:)

      summ(:) = 0.0

   end do

end do

! Update State Variable Q

do inode = 1, nnode
   q(inode,1) = q(inode,1) + dq(inode,1) !+ dqs(inode,1)
   q(inode,2) = q(inode,2) + dq(inode,2) !+ dqs(inode,2)
   q(inode,3) = q(inode,3) + dq(inode,3) !+ dqs(inode,3)
```

```
      q(inode,4) = q(inode,4) + dq(inode,4) !+ dqs(inode,4)
      q(inode,5) = q(inode,5) + dq(inode,5) !+ dqs(inode,5)
      q(inode,6) = gamma * q(inode,5) / q(inode,1)
   end do

   ! Update Residual as the residue from one time step to another

   res(:,:) = dq(:,:)

   ! Switch updated state variables back to primitive form

   call switchpc(nnode, q, gamma, gm1, xgm1, debug, 2)

   if (iter .eq. nstep .AND. debug) then
      endfile(blid)
      close(blid)
   end if

   deallocate(fl, stat = status)
   if (status .ne. 0) stop 'failure to deallocate memory in sweeps'
   deallocate(dfl, stat = status)
   if (status .ne. 0) stop 'failure to deallocate memory in sweeps'
   deallocate(summ, stat = status)
   if (status .ne. 0) stop 'failure to deallocate memory in sweeps'

end subroutine sweeps
```

VITA

Jerry William Carter II received his Bachelor of Science degree in Aerospace Engineering from Texas A&M University in May 2008. He began he graduate study at Texas A&M University in June 2008 and received his Master of Science degree in Aerospace Engineering in May 2010.

H.R. Bright Building, Rm. 701, TAMU-3141
College Station, TX 77843-3141

wcarter@tamu.edu